# AFRL-IF-WP-TR-2001-1560

# A UNIFIED ENVIRONMENT FOR END-TO-END SYSTEM DESIGN

DR. JAMES H. AYLOR
DR. ROBERT H. KLENKE

CENTER FOR SEMICUSTOM INTEGRATED SYSTEMS
UNIVERSITY OF VIRGINIA
THORNTON HALL
CHARLOTTESVILLE, VA 22903-2442

**MARCH 1998**

**FINAL REPORT FOR PERIOD 03 SEPTEMBER 1993 – 23 MARCH 1998**

**Approved for public release; distribution unlimited**

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

# NOTICE

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

Maya Rubeiz

MAYA RUBEIZ, Project Engineer
Embedded Information Sys. Eng. Branch
Information Technology Division
Air Force Research Laboratory

James S. Williamson

JAMES S. WILLIAMSON, Chief
Embedded Information Sys. Eng. Branch
Information Technology Division
Air Force Research Laboratory

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 074-0188*

| | | |
|---|---|---|
| **1. AGENCY USE ONLY (Leave blank)** | **2. REPORT DATE**<br>MARCH 1998 | **3. REPORT TYPE AND DATES COVERED**<br>Final, 09/03/1993 – 03/23/1998 |

**4. TITLE AND SUBTITLE**

A UNIFIED ENVIRONMENT FOR END-TO-END SYSTEM DESIGN

**5. FUNDING NUMBERS**

C: F33615-93-C-1313
PE: 63739E
PR: A268
TA: 02
WU: 05

**6. AUTHOR(S)**

DR. JAMES H. AYLOR
DR. ROBERT H. KLENKE

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

CENTER FOR SEMICUSTOM INTEGRATED SYSTEMS
UNIVERSITY OF VIRGINIA
THORNTON HALL
CHARLOTTESVILLE, VA 22903-2442

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334
POC: Maya Rubeiz, AFRL/IFTA, (937) 255-6653 x3593

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

AFRL-IF-WP-TR-2001-1560

**11. SUPPLEMENTARY NOTES:**

This Report Contains Copyright Material.

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**

Approved for public release; distribution unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** ***(Maximum 200 Words)***

The goal of this project was to create a unified end-to-end design environment which supports the integrated performance and dependability analysis of system level models and has the capability to simulate both uninterpreted and interpreted models in a common simulation environment (mixed-level modeling). This environment provides capability for analysis through simulation or analytical approaches.

The above goal was built into a tool, Advanced Design Environment Prototype Tool (ADEPT), which is functional.

**14. SUBJECT TERMS**

ADEPT, performance analysis, dependability analysis, uninterpreted models, interpreted models, mixed level modeling, petri nets

**15. NUMBER OF PAGES**

386

**16. PRICE CODE**

| | | | |
|---|---|---|---|
| **17. SECURITY CLASSIFICATION OF REPORT**<br>Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE**<br>Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT**<br>Unclassified | **20. LIMITATION OF ABSTRACT**<br>SAR |

**NSN 7540-01-280-5500**

**Standard Form 298 (Rev. 2-89)**
**Prescribed by ANSI Std. Z39-18**
**298-102**

THIS PAGE INTENTIONALLY LEFT BLANK

# TABLE OF CONTENTS

THIS PAGE WAS INTENTIONALLY LEFT BLANK

# 1. Introduction

This document constitutes the final report for contract no. F33615-93-C-1313 awarded under the Rapid Prototyping of Application Specific Signal Processors (RASSP) Technology Base program. This contract was awarded to the Center for Semicustom Integrated Systems (CSIS) at the University of Virginia in August of 1993. The original contract period was 3 years, but CSIS was granted a no-cost extension of one year in 1996. The project involved developing improvements to the CSIS's VHDL-based performance and dependability environment called ADEPT (Advanced Design Environment Prototype Tool). There were four major tasks under this project; Task 1 - develop techniques for reducing the simulation execution time of VHDL-based performance models, Task 2 - develop techniques for performing dependability analysis from VHDL-based performance models, Task 3 - develop techniques for co-simulating and analyzing performance models and lower level behavioral models, and Task 4 - integrate the techniques, models, and tools developed in the tasks above into the deliverable version of the ADEPT tool set.

The remainder of this final report summary is organized as follows; sections 2 and 3 provide the background for the report, including the motivation for the development of the ADEPT design environment, and its organization, sections 4 through 7 summarize the results from Tasks 1 to 4 respectively, section 8 presents some conclusions, and section 9 lists the papers published by UVa researchers working on this project. In addition to this summary, copies of published papers and technical reports that contain additional information and results from each task have been included as appendices to this report. These appendices will be outlined in the relevant sections of this summary

# 2. Background - Motivation

It has been noted by the digital design community that the greatest potential for additional cost and iteration cycle time savings is through improvements in tools and techniques that support the early stages of the design [1]. As shown in Figure 1, decisions made during the initial phases of a product's development cycle determine up to 80% of its total cost. The result is that accurate, fast analysis tools must be available to the designer at the early stages of the design process to help make these decisions. Design alternatives must be effectively evaluated at this level with respect to multiple metrics, such as performance, dependability, and testability. This analysis capability will allow a larger portion of the design space to be explored yielding higher quality as well as lower cost designs.

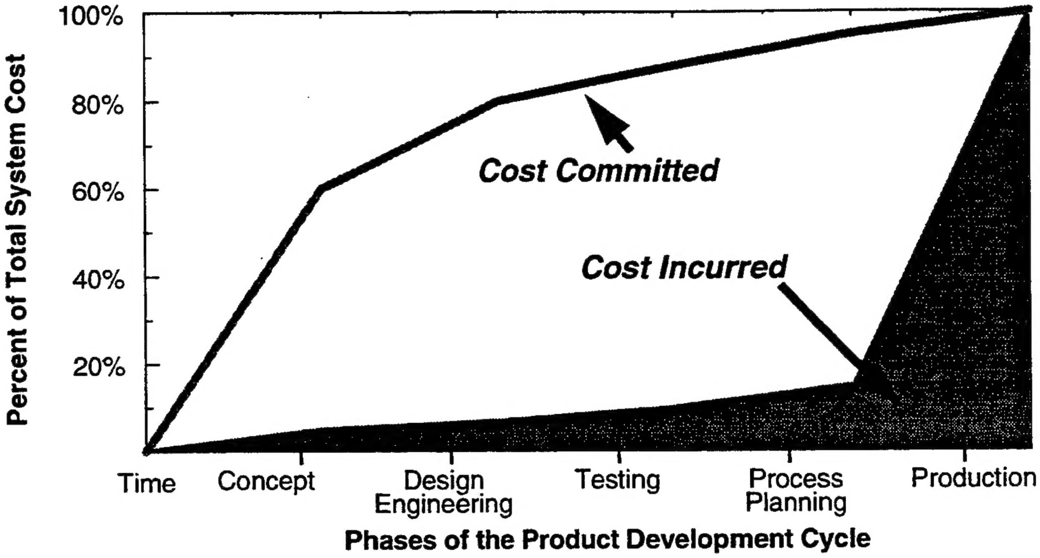


Figure 1. Product costs over the development cycle

There are a number of current tools and techniques that support analysis of these metrics at the system level to varying degrees. A major problem with these tools is that they are not integrated into the engineering design environment in which the system will ultimately be implemented. This problem leads to a major disconnect in the design process where the system level model is developed and analyzed, and then the resulting high level design is specified on paper and thrown "over the wall" for implementation by the engineering design team, as illustrated in Figure 2. As a result, the engineering design team has to interpret this specification in order to implement the system, which often leads to design errors. They also have to develop their own initial "high level" model from which to begin the design process in a top down manner. Additionally, there is no automated mechanism by which feedback on design assumptions and estimations can be provided to the system design team by the engineering design team.

A further problem with existing system level modeling tools is that they force the designers to represent the system using different modeling paradigms for each type of analysis that is to be

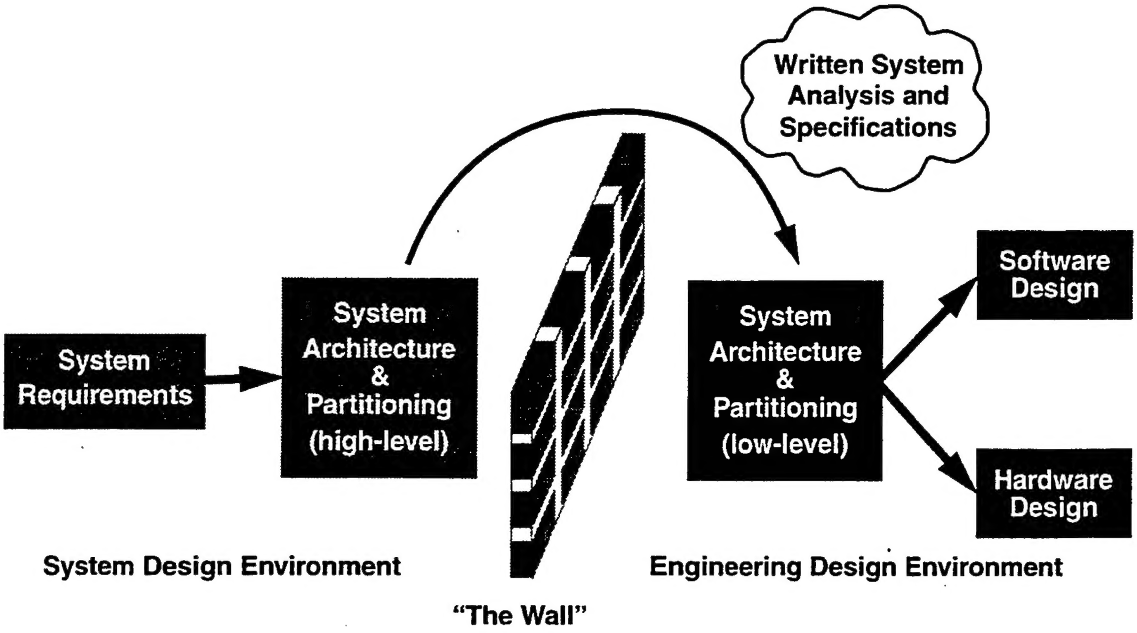

Written System Analysis and Specifications
System Requirements
System Architecture & Partitioning (high-level)
System Architecture & Partitioning (low-level)
Software Design
Hardware Design
System Design Environment
Engineering Design Environment
"The Wall"

Figure 2. The disconnect between system level design environments and engineering design environments

performed. For example, even at the system level, multiple models in different representations are needed to analyze performance and dependability. A great deal of extra work must be done to verify that these models accurately represent the same system. This problem is illustrated in Figure 3. All of these problems could be solved to a large degree if a single system level representation could be used to measure multiple metrics and then be used as a starting point for the engineering design process.

## 3. Background - The ADEPT Design Environment

Two approaches to creating the unified design environment described above are possible. An evolutionary solution is to provide an environment that "translates" data from different models at various points in the design process and creates interfaces for the non-communicating software tools used to develop these models. With this approach, users must be familiar with several modeling languages and tools. Also, analysis of design alternatives is difficult and is likely to be limited by design time constraints.

A revolutionary approach, the one developed under this project, is to use a single modeling

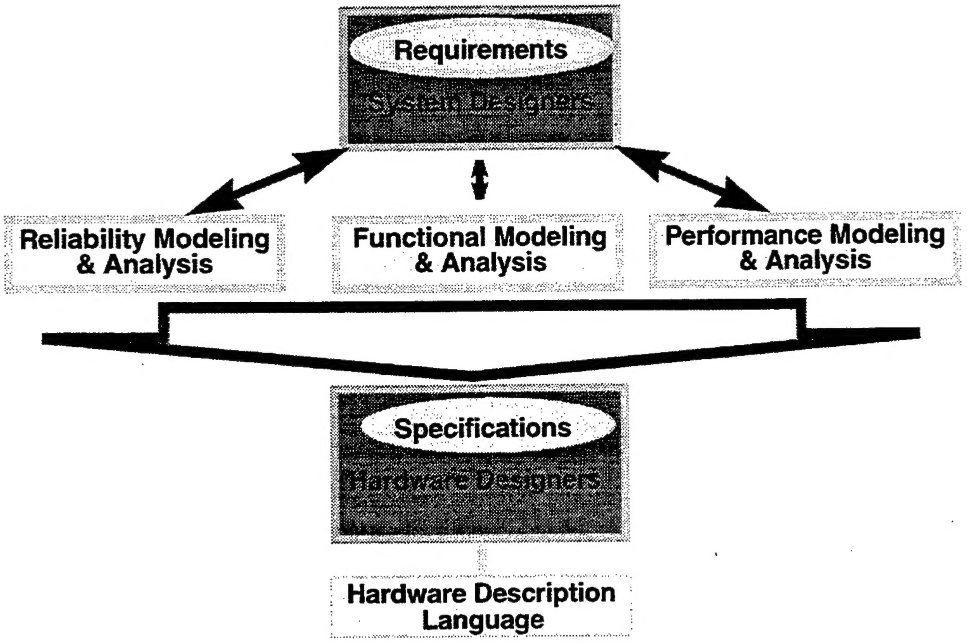

Requirements
System Designers
Reliability Modeling & Analysis
Functional Modeling & Analysis
Performance Modeling & Analysis
Specifications
Hardware Designers
Hardware Description Language

Figure 3. The problem of multiple models required for different analyses

language and mathematical foundation. This approach uses a common modeling language and simulation environment which decreases the need for translators and multiple models, reducing inconsistencies and the probability of errors in translation. Finally, the existence of a mathematical foundation provides an environment for complex system analysis using analytical approaches.

Simulators for hardware description languages accurately and conveniently represent the physical implementation of digital systems at the circuit, logic, register-transfer, and algorithmic levels. By adding a system level modeling capability based on extended Petri Nets and queuing models to the hardware description language, a single design environment can be used from concept to implementation. The environment would also allow for the mixed simulation of both *uninterpreted* (performance) models and *interpreted* (behavioral) models due to the use of a common modeling language.

The main goal of this project was to create a unified end-to-end design environment designed to achieve the goals outlined above. This environment supports the development of system level models of digital systems that can be analyzed for multiple metrics like performance and dependability, and can then be used as a starting point for the actual implementation. A tool called ADEPT (Advanced Design Environment Prototype Tool) has been developed to implement this environment. ADEPT supports both system level performance and dependability analysis in a common design environment using a collection of predefined library elements. ADEPT also includes the capability to simulate both system level and implementation level (behavioral) models in a common simulation environment. This capability allows the stepwise refinement of system level models into implementation level models.

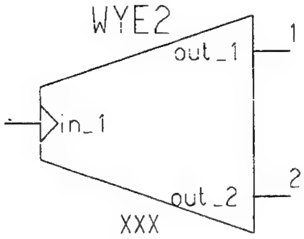
ADEPT implements an end-to-end unified design environment based upon the use of the VHSIC Hardware Description Language (VHDL), IEEE Std. 1076 [2]. ADEPT supports the integrated performance and dependability analysis of system level models and includes the capability to simulate both uninterpreted and interpreted models in a common simulation environment using a technique called *mixed-level modeling*. Mixed-level modeling allows the stepwise refinement of system level models into implementation level models. ADEPT also has a mathematical basis in Petri Nets thus providing the capability for analysis through simulation or analytical approaches [3].

## 3.1 The ADEPT Modules

In the ADEPT environment, a system model is constructed by interconnecting a collection of predefined elements called ADEPT modules. The modules model the information flow, both data and control, through a system. Each ADEPT module has a VHDL behavioral description and a corresponding mathematical description in the form of a colored Petri Net (CPN) based on Jensen's CPN model [4]. The modules communicate by exchanging *tokens*, which represent the presence of information, using a fully interlocked, four-state handshaking protocol [5]. The basic ADEPT modules are intended to be building blocks from which useful modeling functionality can be constructed. In addition, custom modules can be developed by the user if required and incorporated into a system model as long as the handshaking protocol is adhered to. Finally, some libraries of application-specific, high-level modeling modules such a Multiprocessor Communications Network Modeling Library [6] have been developed and included in ADEPT.

ADEPT tokens are implemented as a VHDL record structure. In the token, the two most important fields are the *STATUS* field and the *COLOR* field. The STATUS field is used to implement the token passing mechanism; that is, the "handshaking" between the ADEPT modules. The COLOR field is an array of integers that hold user-specified information. Modules are provided which can manipulate the information in the COLOR field.

An example of an ADEPT module is the *Wye*, shown in Figure 4 with its VHDL behavioral description and its underlying CPN representation. This module models a "fork" construct. When a token arrives at in_1, tokens are placed simultaneously at out_1 and out_2. The input token is not acknowledged (consumed) until both output tokens have been acknowledged.



```
library uvalib;
   use uvalib.uva.all;
   use uvalib.rng.all;

entity wye2 is
   port (in_1 : inout token;
         out_1: inout token;
         out_2: inout token);
end wye2;

architecture ar_wye2 of wye2 is
begin
   pr_wye2 : process (in_1, out_1, out_2)
   begin
      if token_present (in_1)
         and token_removed (out_1)
         and token_removed (out_2) then
         out_1 <= in_1;
         out_2 <= in_1;
      end if;

      if token_acked (out_1) and token_acked (out_2) then
         ack_token (in_1);
         release_token (out_1);
         release_token (out_2);
      end if;

      if token_released (in_1) then
         remove_token (in_1);
      end if;
   end process pr_wye2;
end ar_wye2;
```

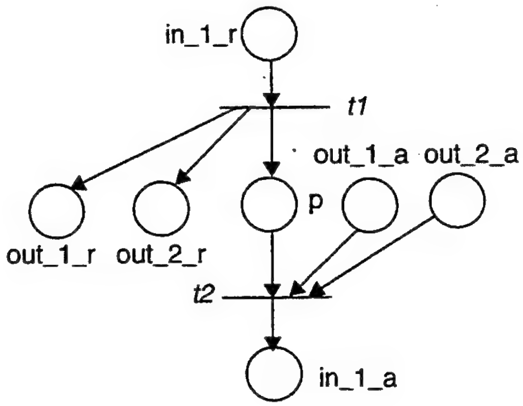


Figure 4. Wye module ADEPT symbol, its behavioral VHDL description, and its CPN representation

In the Petri Net of Figure 4, the places are shown as circles and the transitions are shown as horizontal lines. The places labeled as “xx_x_r” and “xx_x_a” correspond to “ready” and “acknowledge”, respectively. The ready and acknowledge places emulate the handshaking between modules. When a token arrives at the place labeled “in_1_r”, the top transition is enabled, and a token is placed in the “out_1_r”, “out_2_r”, and “p” places. The first two places correspond to a token being placed on the module outputs (out_1 and out_2). Once the output tokens are acknowledged (corresponding to tokens arriving at the “out_1_a” and “out_2_a” places), the lower transition is enabled, and a token is placed in “in_1_a” (corresponding to the input token being acknowledged). The module is then ready for the next input token. Other modules are modeled similarly. The complete CPN descriptions of each of the ADEPT modules can be found in [7].

The set of basic ADEPT modules is divided into six categories: *control* modules, *color* modules, *delay* modules, *fault* modules, *miscellaneous* parts modules, and *hybrid* modules. The control modules are used to manipulate the flow of tokens in a model. A majority of the control modules have been adapted from Dennis [8]. The *Wye* module described above is an example of a control module. ADEPT modules in the color and delay categories enable the manipulation of the token color and model temporal aspects of a system, respectively. The fault modules are used to model the presence of faults and errors in a system model. The miscellaneous modules are modules that perform data collection with the ADEPT system. Hybrid modules aid in the construction of mixed-level models. A more detailed description of the entire ADEPT module set can be found in [9] which is included as an appendix to this report.

## 3.2 The ADEPT Tools

The ADEPT system is available on Sun platforms using Mentor Graphics’ *Design Architect* as the front end schematic capture system, or on Windows PCs using OrCAD’s *Capture* as the front end schematic capture system. The architecture of the ADEPT system is shown in Figure 5.

The schematic front end is used to graphically construct the system model from a library of ADEPT module symbols. Once the schematic of the model has been constructed, the schematic capture system’s netlist generation capability is used to generate an EDIF (Electronic Design

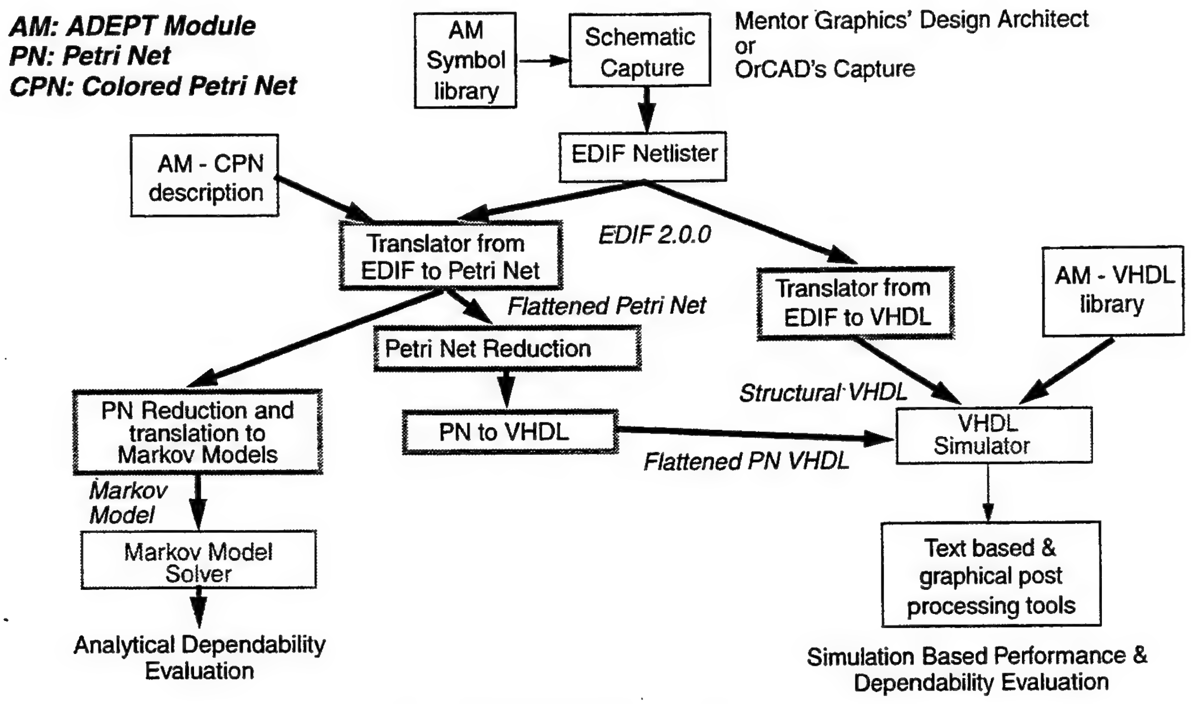

AM: ADEPT Module
PN: Petri Net
CPN: Colored Petri Net
AM Symbol library
Schematic Capture
Mentor Graphics' Design Architect or OrCAD's Capture
EDIF Netlister
AM - CPN description
Translator from EDIF to Petri Net
EDIF 2.0.0
Translator from EDIF to VHDL
AM - VHDL library
Flattened Petri Net
Petri Net Reduction
Structural VHDL
PN Reduction and translation to Markov Models
PN to VHDL
VHDL Simulator
Flattened PN VHDL
Markov Model
Markov Model Solver
Text based & graphical post processing tools
Analytical Dependability Evaluation
Simulation Based Performance & Dependability Evaluation

Figure 5. ADEPT design flow

Interchange Format) 2.0.0 netlist of the model. Once the EDIF netlist of the model is generated, the ADEPT software is used to translate the model into a structural VHDL description consisting of interconnections of ADEPT modules. The user can then simulate the structural VHDL that is generated using the compiled VHDL behavioral descriptions of the ADEPT modules to obtain performance and dependability measures.

In addition to VHDL simulation, a path exists that allows the CPN description of the system model to be constructed from the CPN descriptions of the ADEPT modules. This CPN description can then be translated into a Markov model using well known techniques and then solved using commercial tools to obtain reliability, availability, and safety information.

Figure 6 is an illustration of the construction of a schematic of an ADEPT model using Design Architect. The schematic shown is that of an ADEPT model of a simple three computer system used in the ADEPT tutorial. Most of the elements in this top-level schematic are hierarchical, with separate schematics describing each component. The most primitive elements of the hierarchy are the ADEPT modules.

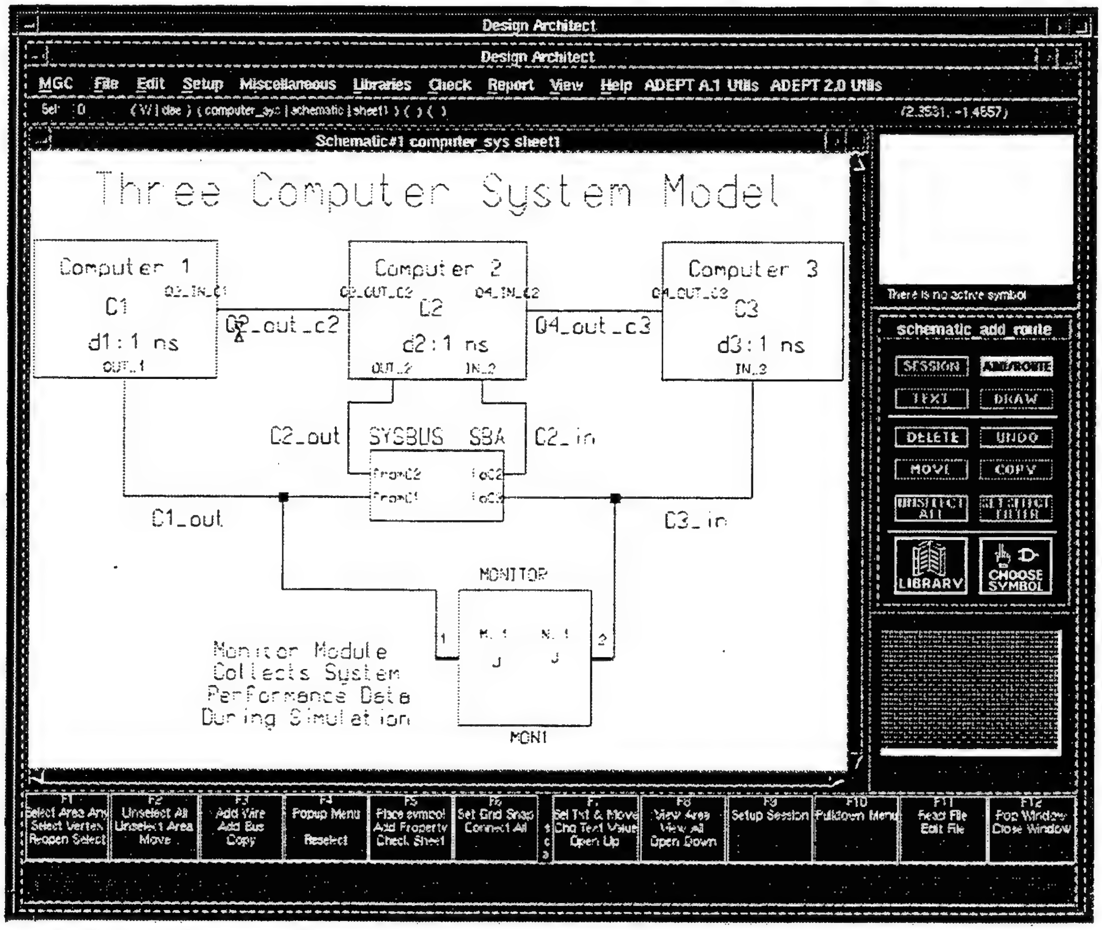
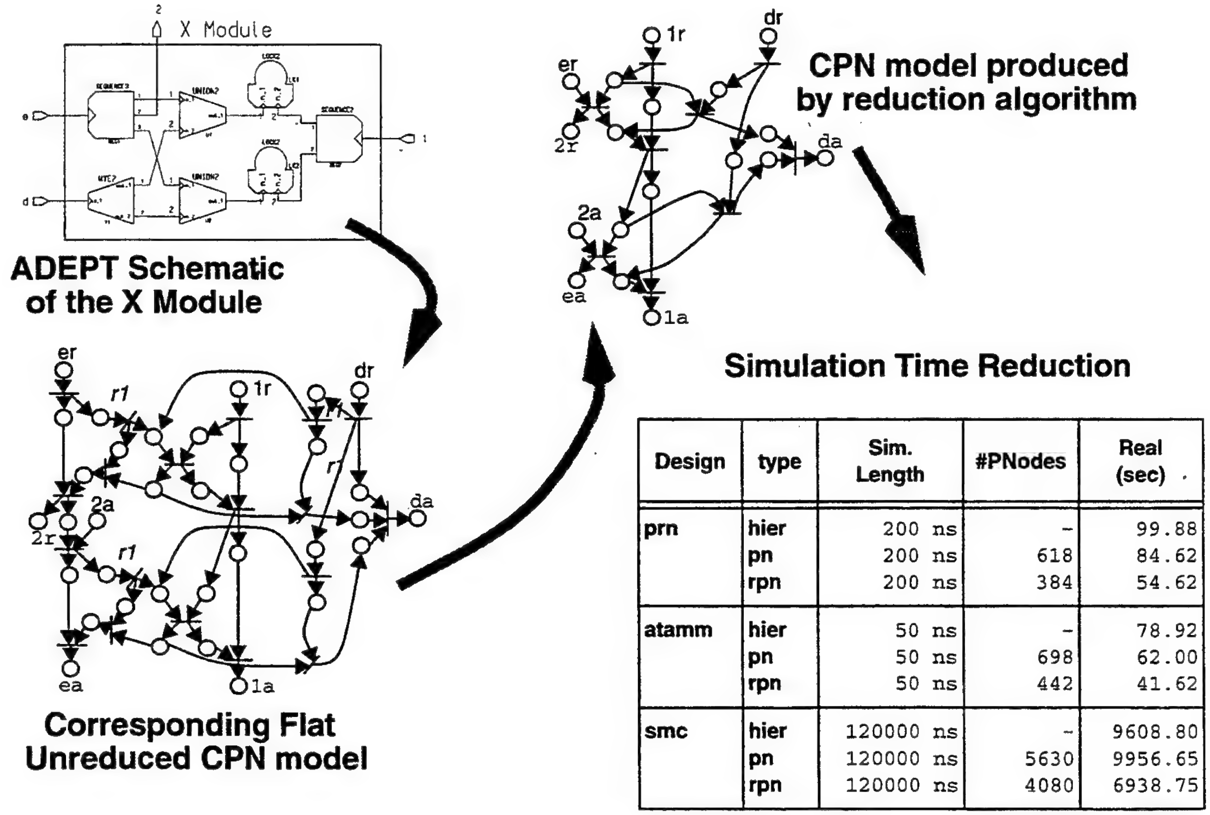


Figure 6. Sample ADEPT schematic (Design Architect)

## 4. Task 1 - Simulation Time Reduction Improvements to ADEPT

Simulation based analysis of a candidate system's performance or dependability can involve running simulations of many different scenarios for long periods of system operational time. Couple this with the desire to evaluate many different candidate system architectures and the adverse impact on the design time of the system due to excessive simulation execution times can be significant. The goal of this task was to develop techniques for reducing the simulation execution times of VHDL-based performance models, specifically ADEPT models. Techniques for reducing the simulation execution time concentrated on two different approaches, using the Petri Net foundation of ADEPT to reduce the overall complexity of the ADEPT model, thereby reducing its simulation execution time, and developing VHDL coding styles for ADEPT models that reduce simulation execution time.

Recall that as shown in Figure 5, ADEPT models can be translated into Colored Petri Net models from their EDIF netlist format. This is performed by substituting each ADEPT module in a system model with its CPN representation. Once this flat, CPN version of the overall ADEPT model is produced, it can be reduced by eliminating redundant places and transitions using a set of 5 reduction rules developed specifically for this purpose. The reduced CPN model can then be translated into VHDL and simulated. This simulation will produce the same results in terms of performance analysis of the system being modeled, but because the model was reduced, the simulation time will also be reduced. This process of reducing the simulation time of ADEPT models by reducing the corresponding Petri Net representation is shown in Figure 7.


X Module
ADEPT Schematic of the X Module
er
r1
1r
dr
2a
2r
da
ea
1a
Corresponding Flat Unreduced CPN model
CPN model produced by reduction algorithm

**Simulation Time Reduction**

| Design | type | Sim. Length | #PNodes | Real (sec) |
|---|---|---|---|---|
| prn | hier | 200 ns | – | 99.88 |
| | pn | 200 ns | 618 | 84.62 |
| | rpn | 200 ns | 384 | 54.62 |
| atamm | hier | 50 ns | – | 78.92 |
| | pn | 50 ns | 698 | 62.00 |
| | rpn | 50 ns | 442 | 41.62 |
| smc | hier | 120000 ns | – | 9608.80 |
| | pn | 120000 ns | 5630 | 9956.65 |
| | rpn | 120000 ns | 4080 | 6938.75 |

Figure 7. Petri Net reduction example and results

Although the Petri Net reduction technique described above resulted in a typical reduction of the simulation execution time of an ADEPT model of approximately 30%, it suffers from two major disadvantages. First, it requires that each ADEPT module in the model have a corresponding CPN representation, which makes the addition of new modules to the ADEPT library much more difficult. This is especially true of high level modules for modeling systems such a multiprocessors and networks which have complex behavioral functionality, even in a high-level model, and thus have a complex Petri Net representation which is difficult to develop. Second, although the Petri Net simulation is guaranteed, by construction, to produce the same results as the behavioral VHDL ADEPT model, it is still difficult for the designer to follow the internal workings of the Petri Net model which makes debugging more difficult and time consuming.

In order to alleviate these problems while also attacking the problem of excessive simulation times, it was decided to attempt to reduce the simulation time of the normal ADEPT VHDL models directly. A detailed study was made of the factors that cause simulations of ADEPT VHDL models to take excessive simulation execution time [10,11]. It was discovered that the size of the token signal had a large influence on the simulation execution time. The size of the token was found to be a problem because the VHDL bus resolution function used to implement the four state handshaking protocol has to copy in the entire token record structure despite the fact that it only needs to deal with the token's STATUS field, not the token's COLOR field. In the original version of ADEPT, the tokens were of fixed size in terms of the number of tag fields. Many models however, do not use all of the tag fields available. Therefore, in order to decrease simulation times for ADEPT models, a new version of ADEPT was created that allows the user to select a smaller token for use in his or her model. In addition, it was found that eliminating the bus resolution function altogether, and using two uni-directional signals to implement the four-state token handshaking, reduced the simulation time even further. However, this "two wire" system has the significant drawbacks of increasing the complexity of model construction and more difficult interpretation of a token signal's state during simulation. An experimental version of ADEPT where the token is actually split into two VHDL signals, a separate STATUS signal and COLOR signal, has been developed. Splitting the token into two signals allows the VHDL bus resolution function to deal only with the token STATUS field, decreasing simulation execution time almost to that of the "two wire" system. Results in terms of simulation time for the simulation of a large ADEPT modeling benchmark called ATAMM, for various token sizes, with a bus resolution function (across the complete token, STATUS and COLOR) and with two wire handshaking, is shown in Figure 8.

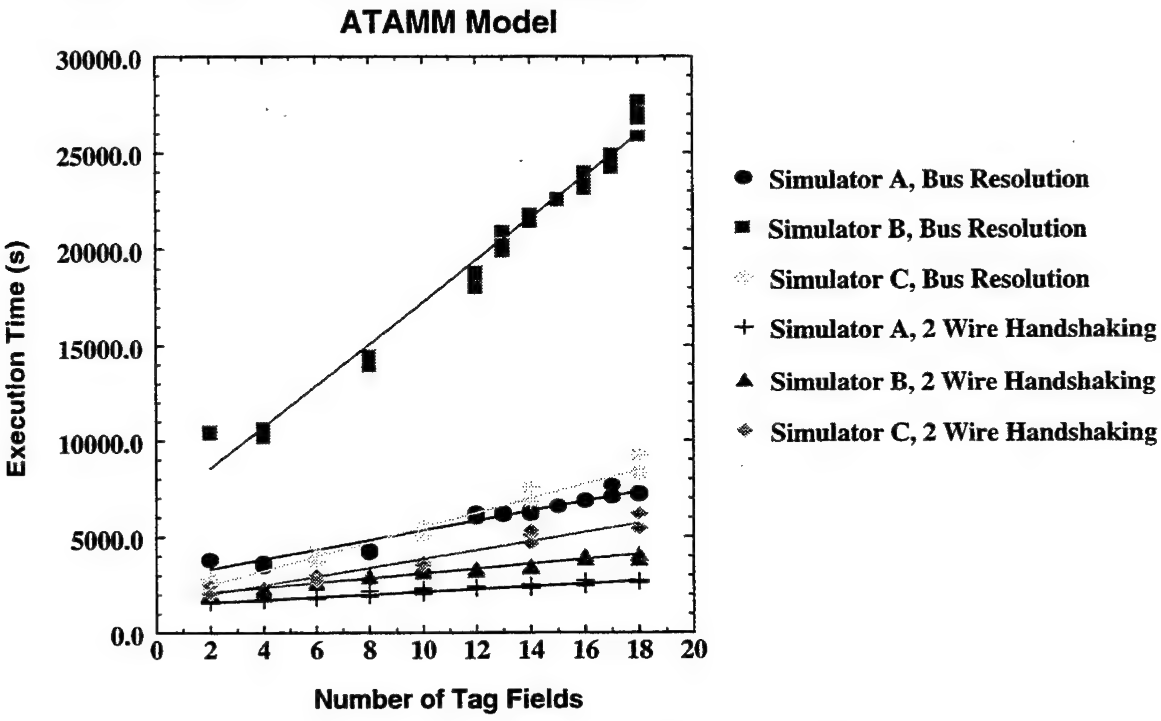
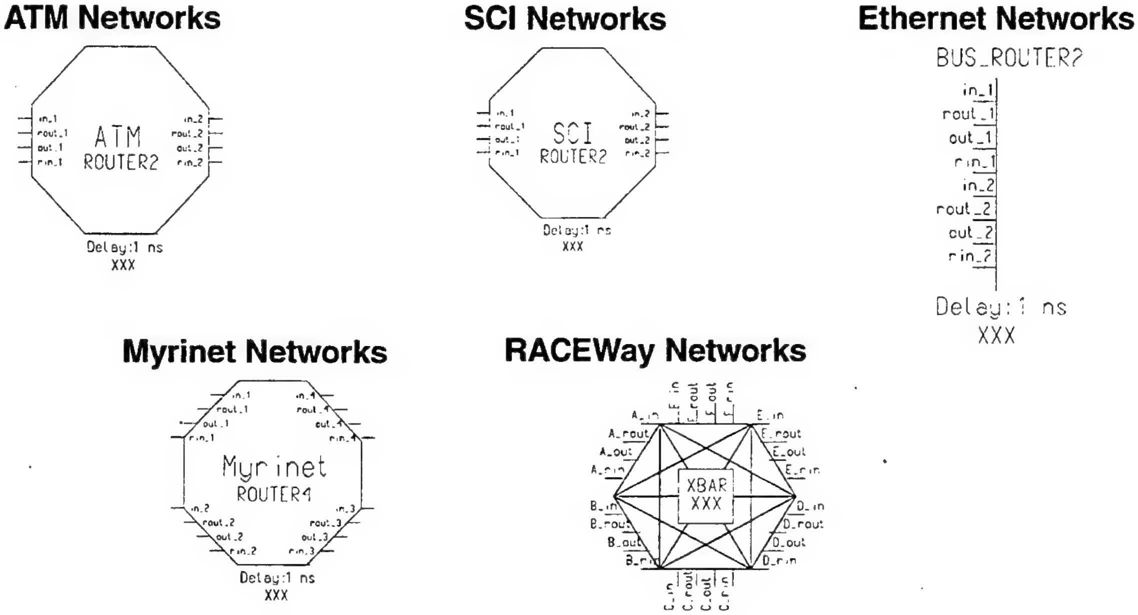


Figure 8. Simulation time reduction results

In addition to the execution times for simulation of ADEPT models, it was discovered that construction of ADEPT models has taken a longer period of time because of the low level of functionality of the basic ADEPT building blocks. In order to address this problem, several libraries of modeling elements for specific application areas have been developed. Among those are a Task Level Modeling Library intended to model applications at a very high level similar to queueing models, a Cycle Based System Modeling Library for modeling synchronous systems such as microprocessors from a high-level down to the RTL level, and a Multiprocessor Communications Network Modeling Library. This latter library is intended to model embedded multiprocessor systems such as those used in the RASSP program. It includes network routers that model the ATM, SCI, Ethernet, Myrinet, and Mercury RACEWay communications protocols. It also includes a simple CPU element that models a compute-send-receive type level of abstraction. The network routing elements from this library are shown in Figure 9. More information on this modeling library and its application can be found in [6] and [11].


**ATM Networks**

**SCI Networks**

**Ethernet Networks**

**Myrinet Networks**

**RACEWay Networks**

Figure 9. The Multiprocessor Communications Network Modeling Library elements

Included as appendices to this report are several published papers which contain further information on the simulation time reduction work accomplished under this task. The paper entitled “A VHDL Based Environment for System Level Design and Analysis” by Swaminathan, et. al., contains more information on the Petri Net based model reduction techniques as well as more information on the Petri Net based dependability analysis techniques described in the next section. The paper entitled “The Analysis of Modeling Styles for System Level VHDL Simulations” by Voss, et. al., contains more information on the effects of token size reduction and bus resolution functions on the simulation execution time of ADEPT models. Finally, “Performance Modeling of Multicomputer Systems in VHDL using ADEPT” contains more information on the Multiprocessor Communications Network Modeling Library added to ADEPT and examples of its use in modeling several types of multiprocessor systems.

## 5. Task 2 - Dependability Modeling Improvements to ADEPT

The goal of this task was to further develop the capability in ADEPT for performing dependability analysis in the same framework and using the same models as performance analysis. There were two main areas where work under this task was concentrated, developing the capability for performing dependability analysis of system-level models using Petri Net to Markov model conversion, and developing an ADEPT-based environment for hardware/software codesign and analysis of dependable systems.

### 5.1 High Level Dependability Analysis

System-level dependability analysis is supported in ADEPT by the CPN descriptions of the ADEPT modules. A system-level ADEPT model can be converted into a CPN description by replacing each module with its CPN description. The CPN model is then reduced using reduction rules developed for dependability analysis [12] and converted to a Markov model using techniques similar to those described in [13]. The Markov model can then be solved to generate dependability metrics using well known techniques and tools.

Dependability analysis using this method is illustrated by an example of a Triple Modular Redundant (TMR) with a Spare system. An ADEPT schematic of the TMR with a Spare system is shown in Figure 10. In this example, it is assumed that there is some form of fault detection in processor P3 that disconnects P3 when it fails and brings processor P4 on-line and that the coverage factor for detecting failures in P3 is 1 (100%), although other values could be used.

The CPN model of each processor module is shown in Figure 11a and is obtained by replacing each ADEPT module by its corresponding reduced CPN definition. Rules used to reduce the CPN in Figure 11a to the CPN in Figure 11b are also illustrated. The remaining components of the system are reduced in a similar fashion and are combined to obtain the reduced CPN representation of the complete system as shown in Figure 11c. The corresponding Markov model is also shown in Figure 11d. The important point here is that the Markov model is constructed from the ADEPT model using automated techniques. The designer does not need to build an additional model in order to gain reliability information. ADEPT also supports reliability analysis using simulation of the system level ADEPT models [14]. Figure 12 shows the results obtained

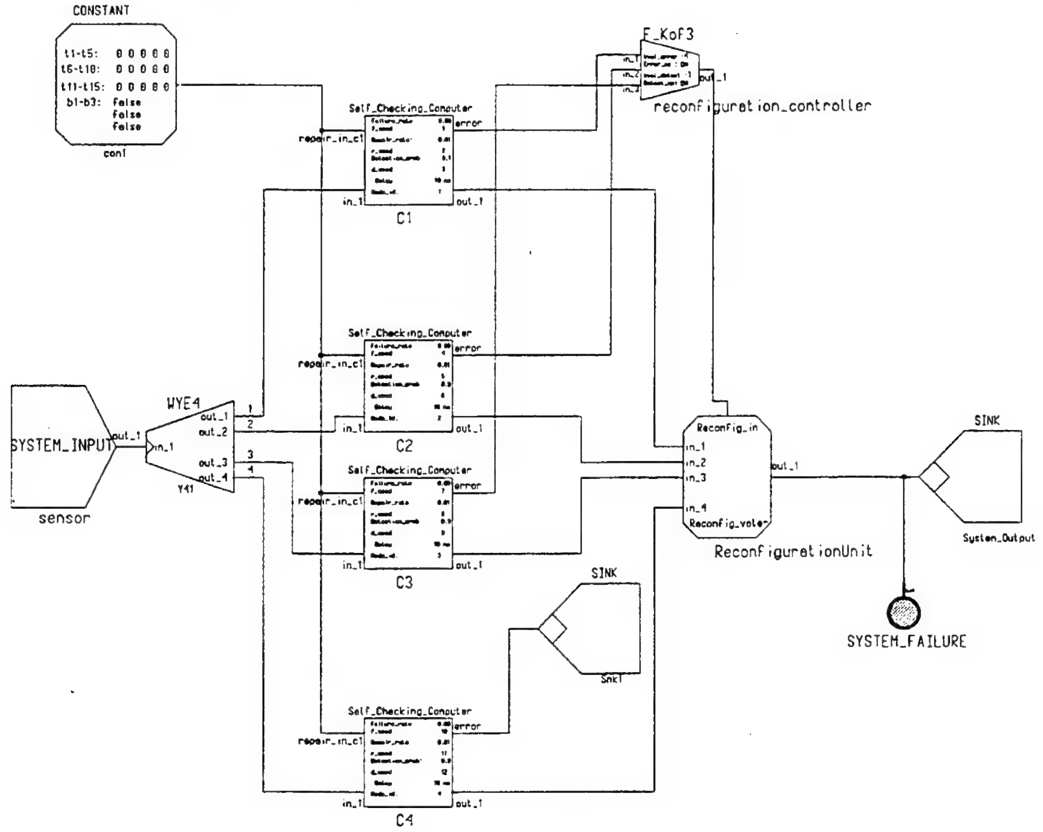


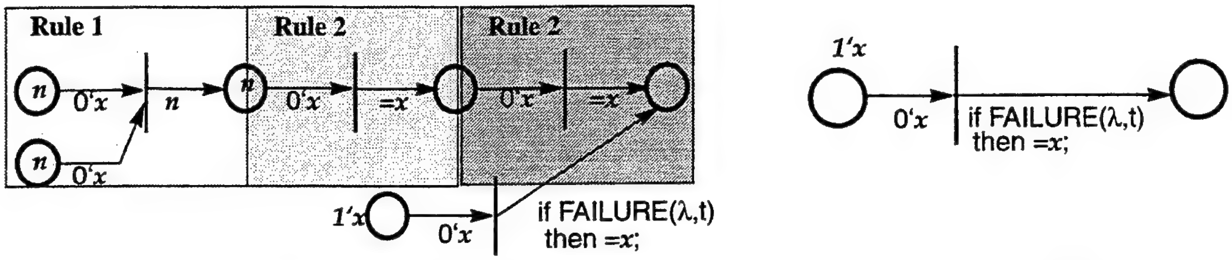
Figure 10. TMR with a spare schematic

from dependability analysis of the TMR with Spare system in the form of reliability and safety measures versus mission time in hours.

## 5.2 Dependable System Codesign

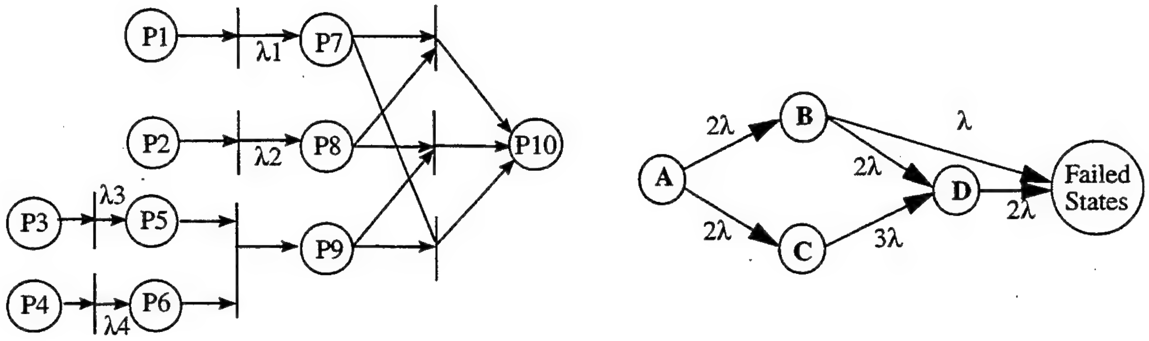
Although the process described above provides the ability to perform integrated performance and dependability analysis on high level models of systems, there is some difficulty inherent in constructing models of dependable systems using the standard ADEPT modules. These types of dependable systems includes both hardware and software constructs such as voters, checkpointing and rollback mechanisms, and watchdog timers.

In order to address this problem, an integrated design and analysis environment for dependable systems was developed based on ADEPT. Dependable system codesign using this environment is illustrated in Figure 13.



**(a) Corresponding CPN Model of a processor**

**(b) Reduced CPN Model of a processor**



**(c) System CPN model**

**(d) System Markov model**

Figure 11. TMR with spare model reduction

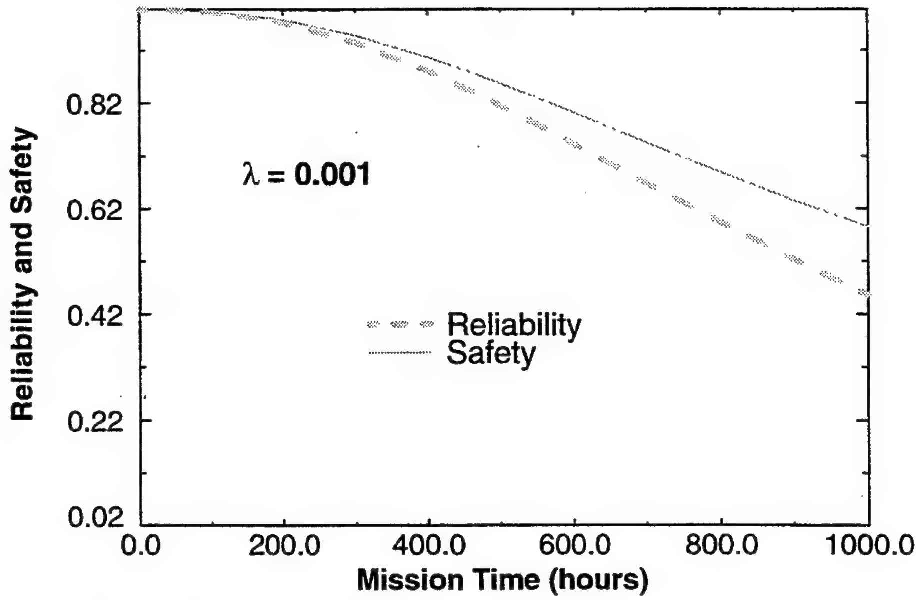


Figure 12. TMR with a spare dependability analysis results

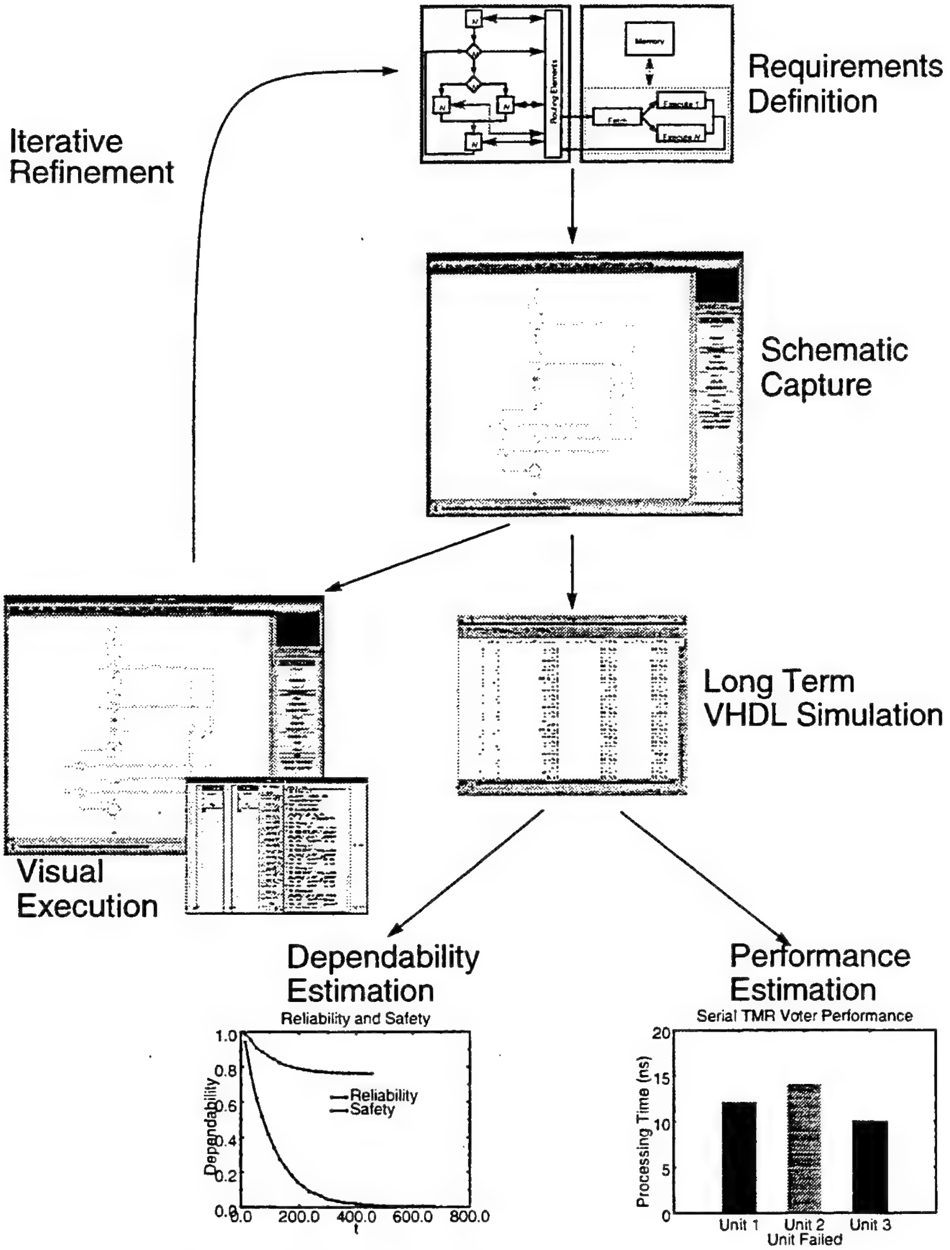


Figure 13. Dependable system codesign flow

The process begins with the construction of an ADEPT model using the library of modeling modules created expressly for modeling dependable hardware/software systems. The model can then be simulated and the results viewed interactively using the AnimateADEPT tool described in section 7. Statistical results of long-term simulations can also be gathered and analyzed by dependability and performance metric display tools created specifically for this environment.

The hardware/software system models created for this dependability analysis environment follow the request/resource modeling paradigm presented in [15]. An example of this modeling paradigm used to model the execution of the algorithm for $b^2$-$ac$ is shown in Figure 14. Computation of this function is accomplished by the appropriate movement of tokens through the software graph representing the performance of the required computations. Each node in the software graph represents a computation has an associated request node. The request nodes model the performance of the computation on actual hardware by sending a request for resources to the hardware model. The hardware resource then incurs a delay and responds back that the request has been serviced. The software graph can then continue execution until the next computation is reached. The interface between the hardware and software graph schedules the requests onto the resources that are available. The ordering and binding of requests to specific resources is determined by the user. Because time is a physical property and software is an informational quality, time is not specified in the software graph, or to be more specific, no duration is specified for each computation node in the software graph. Time is accounted for by delaying the response of the hardware to requests made to it by the software graph.

This unified representation for hardware and software allows different allocations of computations to hardware and software to be easily modeled. It also allows the easy insertion of modeled faults into either the software or hardware systems using the same mechanism. This allows analysis of the dependability of the combined hardware and software system. A library of elements designed for modeling dependable systems using this modeling paradigm has been developed. It uses the dataflow model of computation and includes elements to model fault insertion and detection as well as dependable software constructs such as checkpointing and rollback.

Two papers which describe the dependability modeling aspects of ADEPT developed under this project are included as appendices to this final report. The first paper, "Integrated Performance and Dependability Analysis Using the Advanced Design Environment Prototype Tool (ADEPT)" by Rao, et. al., describes the Petri Net based dependability analysis capabilities of ADEPT in more detail as well as the ADEPT-Rest interface - a simulation based dependability analysis capability added to ADEPT under funding from NASA. The second paper, "Dependable

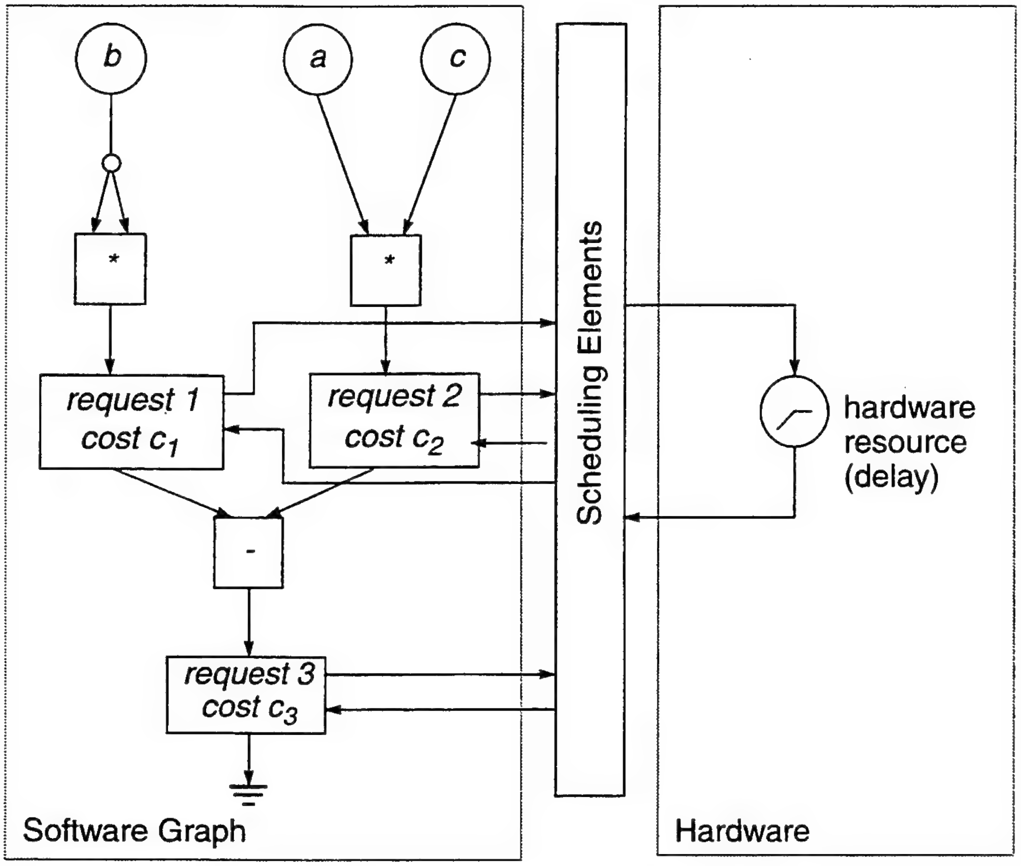


Figure 14. Codesign model of $b^2 - ac$

System Codesign using Data Flow Models" by Choi, et. al., describes the dependable systems codesign environment in more detail.

## 6. Task 3 - Mixed-Level Modeling Improvements to ADEPT

The goal of this task was to extend the mixed-level modeling capabilities of ADEPT. As stated previously, mixed-level modeling is the capability to co-simulate uninterpreted (system level) and interpreted (behavioral) models in a common simulation environment. Unlike uninterpreted components, interpreted components contain functionality responsible for mapping values at their inputs to values at their outputs and typically contain more detailed timing and event granularity. As system components are refined to the interpreted level, it is very beneficial to simulate their models in the context of the entire system. In order to be able to perform this refinement in an incremental fashion, the capability to co-simulate uninterpreted and interpreted components in the same model is required.

When constructing a mixed-level model, an interface must be placed between the uninterpreted and interpreted components as shown in Figure 15. The interface consists mainly of the Uninterpreted to Interpreted (U/I) operator and the Interpreted to Uninterpreted (I/U) operator. Note that the goal of the interface is to have the interpreted element, along with its U/I and I/U interfaces, behave the same as the uninterpreted element it replaces with the exception of providing more detailed and accurate timing and functional information. This means that the interface must accept a token (or tokens) from the uninterpreted model, apply the proper inputs to the interpreted component, and then according to the outputs from the interpreted component, release the token(s) back to the uninterpreted model.

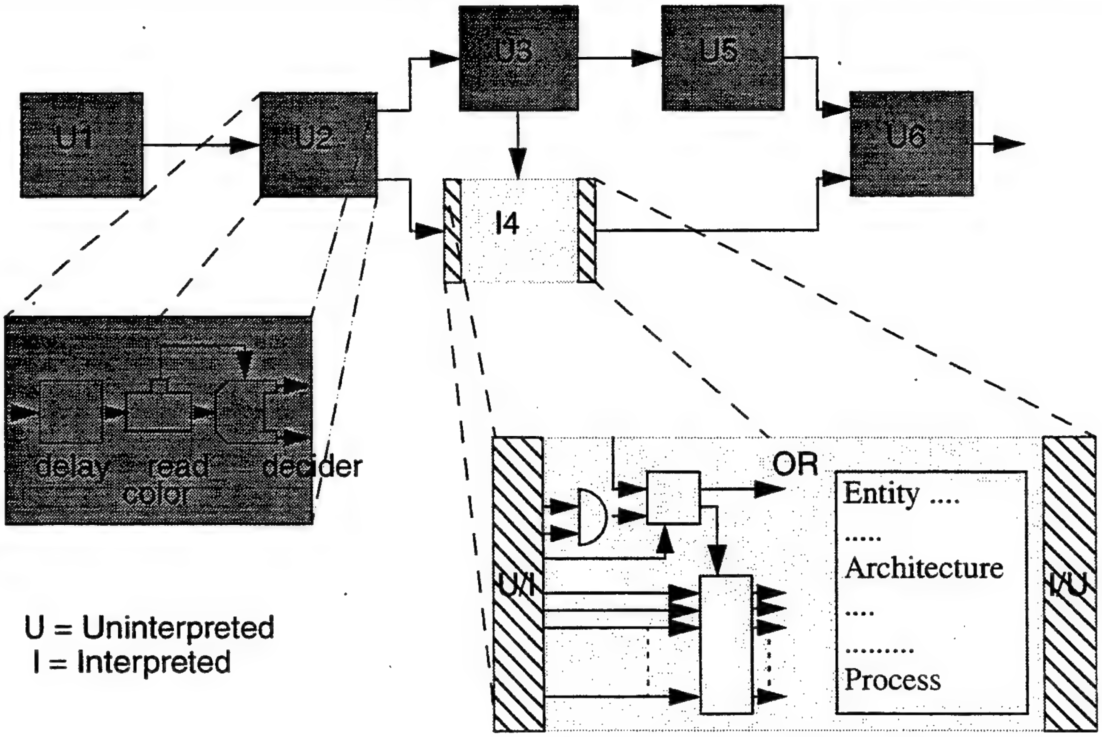


Figure 15. The general structure of the mixed-level modeling interface

In behavioral models, signals are usually of a less abstract data type than tokens (such as bit, std_logic, integer, or real) and must have actual values associated with them in order for the model to function correctly. In addition, behavioral models usually resolve timing events to a finer granularity than uninterpreted models. The mixed-level modeling interface must therefore resolve these differences in timing and data abstraction between the uninterpreted and interpreted modeling domains.

Obviously there are a number of factors that influence the timing and data abstractions that must be resolved by the mixed-level interface. Among these factors are the type of uninterpreted model that the interpreted component is being inserted into, the type of interpreted component, either combinational or sequential, the interpreted component's complexity, and the objective of the mixed-level model, either timing verification or functional verification.

In order to classify these factors and their effect on the mixed-level interface, a taxonomy of mixed-level models was created in concert with researchers at the Honeywell Technology Center [16]. An illustration of this taxonomy is shown in Figure 16. Notice that general mixed-level models are broken down by the type of system model, the type of interpreted element, and the modeling objective as stated previously. Most of the research efforts in mixed-level interfaces in ADEPT have been aimed at the goal of timing verification. A methodology and library elements to construct mixed-level interfaces for timing verification of combinational interpreted elements has been developed for ADEPT under previous research [17]. Current efforts in mixed-level interfaces has concentrated on timing verification for sequential interpreted elements. Sequential interpreted elements are further broken down into sequential datapath elements (SDE) and sequential control elements (SCE). SCEs are sequential elements that are simple finite state machines (FSMs). SDEs are sequential elements that include a controller, in the form of an FSM, and its associated datapath. They are also referred to as FSMDs (Finite State Machines with Datapaths). Examples of SDEs include simple floating point coprocessors or application specific coprocessors like an FFT chip. Note that since SCEs are actually a simpler subset of SDEs, the mixed-level interfaces and methodologies that have been developed for SDEs will also work with SCEs.

Mixed-level interfaces for two general categories of SDEs have been developed for ADEPT, those SDEs that can be described as FSMs in terms of their Sate Transition Graph (STG) and those that are too complex to represent using an STG. Examples of the latter include general

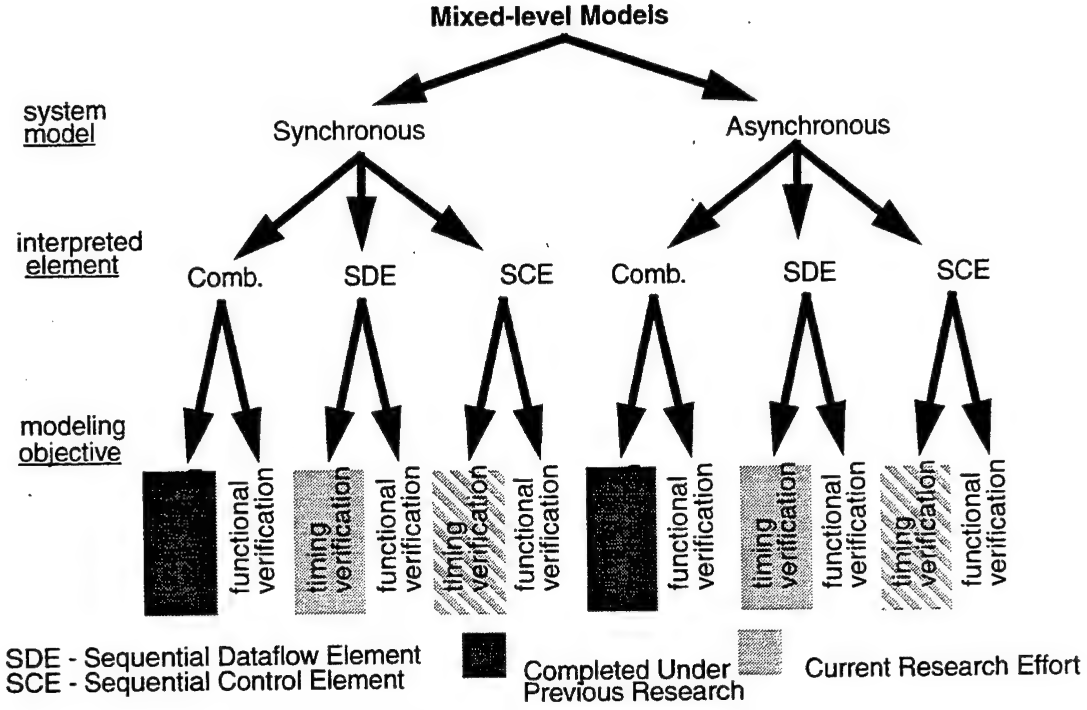


Figure 16. The mixed-level modeling taxonomy

purpose microprocessors or DSPs, and complex floating point coprocessors. These interfaces will be discussed in more detail in the next two sections.

## 6.1 Mixed-Level Modeling for SDE Interpreted Components

As stated previously, the mixed-level interface must resolve the problems of timing and data abstraction between interpreted and uninterpreted components. For sequential interpreted components, the timing abstraction problem is one of supplying a clock signal to the interpreted component, and determining when to release a token to the uninterpreted model based on the outputs from the interpreted component. Solving the data abstraction problem involves determining the values to be placed on the inputs to the interpreted component when a token arrives. The arriving token may contain partial information, such as the operation the sequential component must perform, but other portions of the inputs, such as the data values, may be unknown. In order to gain valuable information, such as best-case or worst-case delays, from the mixed-level model, these unknown inputs must be supplied proper values.

In the mixed-level interface, it is the function of the U/I operator to drive the input signals to the interpreted element according to information carried within the input token, or if necessary, by deriving it. The I/U operator's function is to release tokens at the appropriate time, possibly with new values according to the output signals of the interpreted element. Together, the U/I and I/U operator solve the timing abstraction problem.

The structure of the mixed-level interface developed to perform these functions is shown in Figure 17. The U/I operator is composed of the following building blocks: Driver, Activator and Clock-Generator. The I/U operator is composed of an Output_Condition_Detector, a Colorer and a Sequential_Releaser.

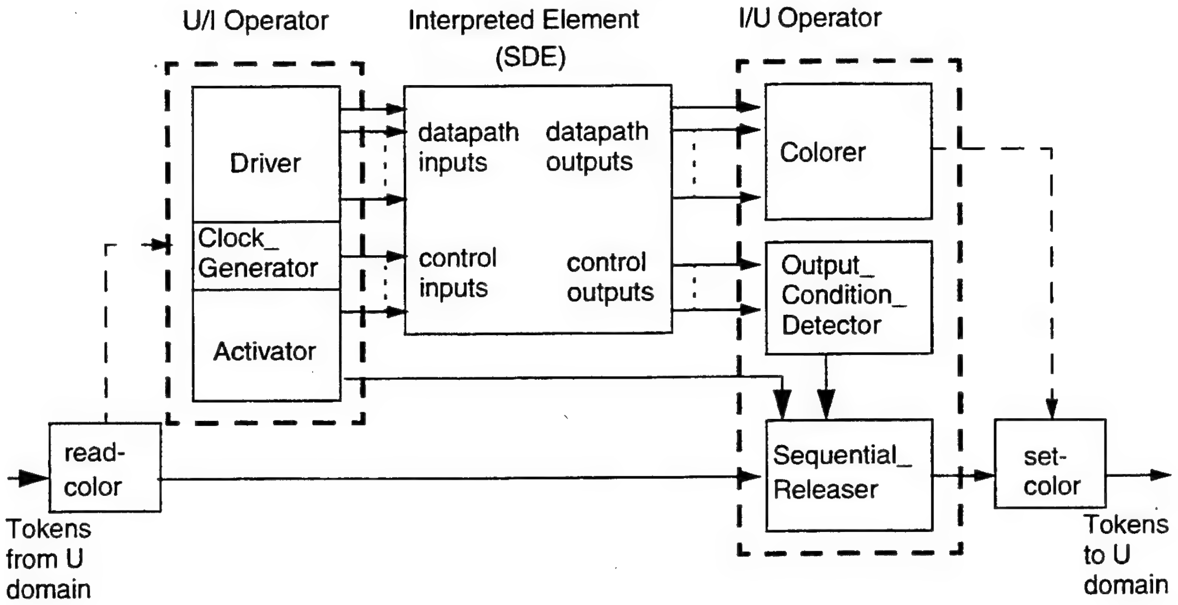


Figure 17. The SDE mixed-level interface structure

In the U/I operator, the Activator is used to signify the arrival of a new token to the interpreted element. The Driver is used to read information from the token's tags and to drive the datapath input signals (and potentially some control inputs as well) to the interpreted element according to predefined assignment properties. The Driver also derives the proper values to drive on the inputs that are not provided in the input token (called the unknown inputs) using the methodology described below. The Clock_Generator generates the clock signal for the interpreted sequential element.

In the I/U operator, the Output_Condition_Detector is used to signify the completion of the interpreted element data processing operation, by comparing the control outputs to predefined values. This process is based on the typical feature of an FSMD, which indicates completion of data processing by asserting some of its output signals. The Colorer samples the datapath outputs and maps them to color fields according to predefined binding properties. The Sequential_Releaser, which "holds" the original token, releases it back to the uninterpreted model upon receiving the signal from the Output_Condition_Detector. The information carried by the token is then updated by the Colorer and the token flows back to the uninterpreted part of the model.

In terms of the data abstraction problem, a methodology has been developed to use the behavioral description of the sequential interpreted element to assist in determining the proper values to be placed on the unknown inputs. The behavioral description of the sequential interpreted element is in the form of the FSMD's State Transition Graph (STG). The STG is a directed graph in which the nodes are the states which the SDE component can attain, and the edges are directed arcs which are marked with the input combinations which move the SDE along them, and the resulting outputs from the SDE for that state transition.

The methodology is designed to utilize the STG to determine the proper sequence of values to apply to the unknown inputs of the SDE such that the longest (worst-case delay) or shortest (best-case delay) path is taken from the starting state to the state which indicates completion of processing. Graph theoretic algorithms are used to reduce the STG by removing state transitions that are caused by inputs that do not influence the delays through the SDE, and searching the resulting STG for the longest or shortest path. The methodology has four steps:

- Determine the outputs and associated values from the SDE that signify the completion of processing data,
- Minimize the STG to remove those inputs that do not influence delay,
- Search the resulting STG for the longest (shortest) path from the initial state to the final state, and
- Use the resulting delay to determine the token release time.

Results of an example mixed-level model with an SDE interpreted component are shown in Figure 18. This example consists of a high-level performance model of microprocessor with a fetch unit and a separate integer and floating point execution unit. A fully behavioral description of the floating point unit was inserted into the performance model using the interface and methodology described above. As can be seen from the graph, an upper bound and lower bound on performance of the system was generated by filling in the unknown inputs to the interpreted component appropriately. Note that the X axis is the fraction of total number of inputs to the interpreted component that had known values (from the tokens in the performance model) associated with them. This fraction increases as the performance model is refined and more detail is added to it. As refinement increases, the bounds on the performance given by the mixed-level model converge to a final value which is the actual result. Notice that the initial estimate from the performance model was slightly different from the actual value produced by the fully refined mixed-level model. More detail on the methodology for solving the data and timing abstraction problem for SDE components and example results can be found in [18].

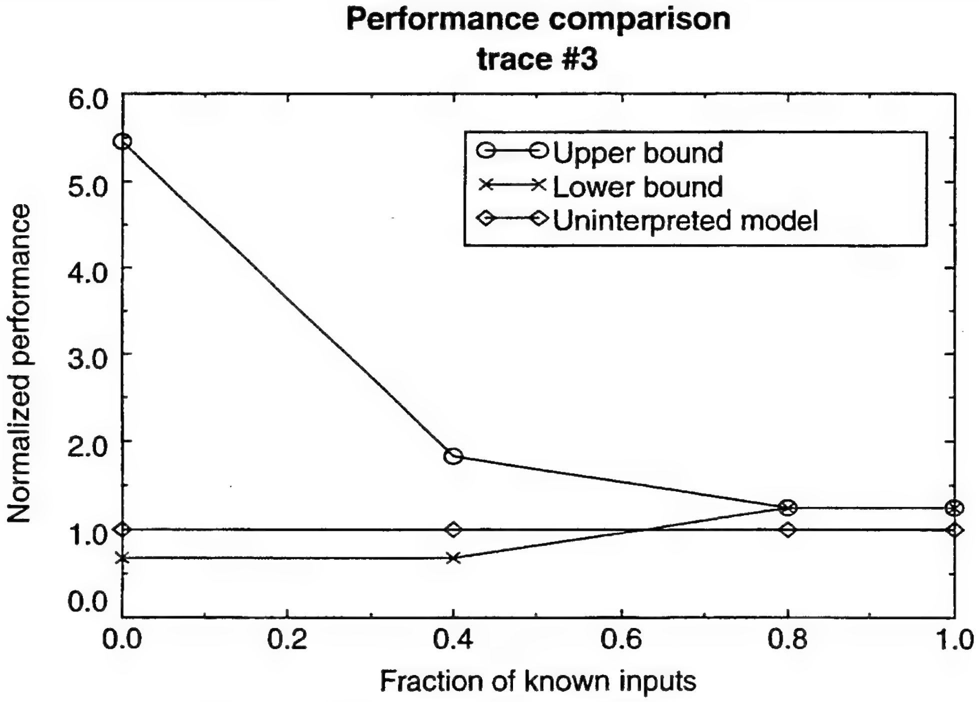


Figure 18. Performance vs. fraction of known inputs

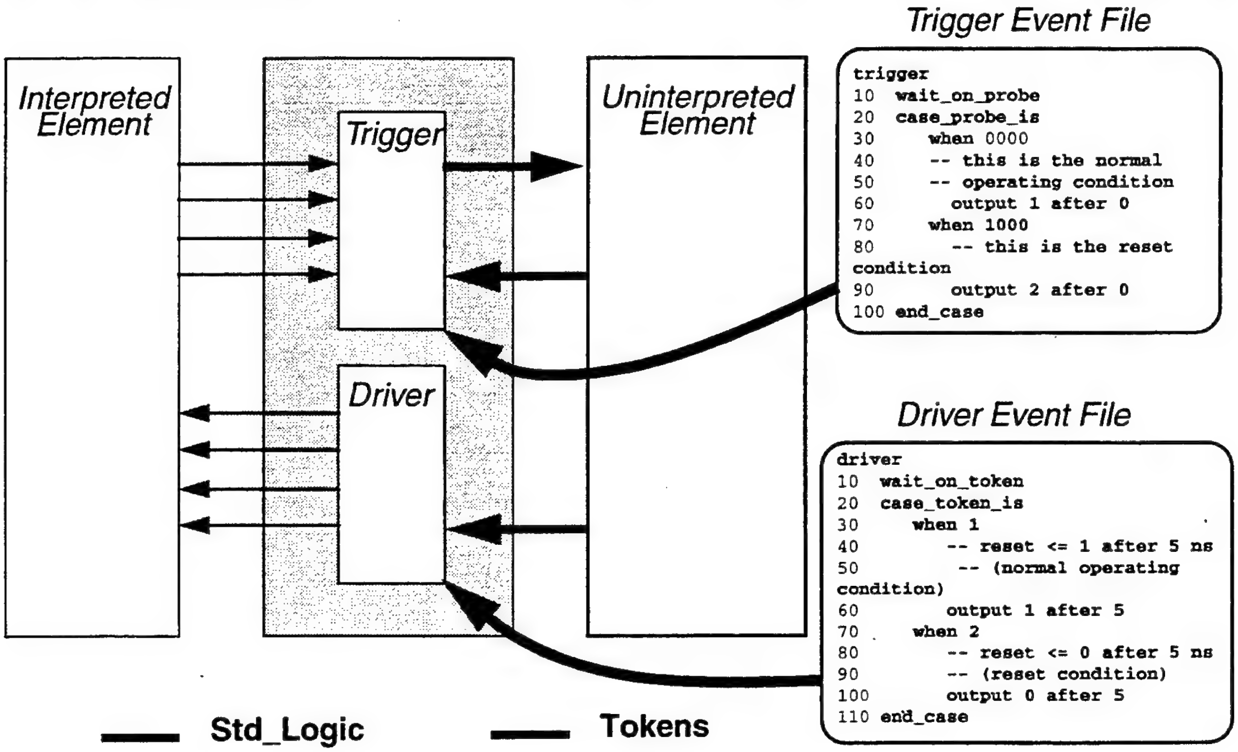
## 6.2 Mixed-Level Modeling for Complex Sequential Components

The methodology and interface elements described above allow the use of sequential interpreted components in mixed-level models. The use of the STG to determine best and worst case delays for timing verification is a powerful methodology. However, many useful mixed-level models can be constructed that include sequential interpreted components that are too complex to be represented as FSMs, such as microprocessors, floating point coprocessors, etc. For these types of complex sequential interpreted components a generalized, flexible mixed-level interface called the watch-and-react interface was created.

The two main elements in the watch-and-react interface are the *trigger* and the *driver* as shown in Figure 19. Both elements have ports that can connect to signals in the interpreted components of a model. Collectively, these ports are referred to as the *probe*. Each element also has an associated program file which the user writes to program the elements for the specific application. The trigger and driver programs are written in a special interpreted language that has some similar constructs to VHDL.

The trigger's function is to monitor the important signals of the interpreted component and generate associated tokens for the uninterpreted model. The driver's function is to receive tokens from the uninterpreted model and generate the required values on the inputs to the interpreted component. Together these two modules allow the user to construct a mixed-level interface that solves the timing abstraction problem in a systematic fashion. Although the driver can be used to drive values on the inputs of the interpreted component, because these types of interpreted components are too complex to analyze using their STGs, the user must solve the data abstraction problem and specify the values to be used in a more ad hoc fashion.

The watch-and-react interface provides a methodology for constructing mixed-level interfaces for complex interpreted elements without requiring the user to develop his or her own interface elements. The use of a programming language in the trigger and driver maintains maximum flexibility and the fact that the language is interpreted means that the ADEPT VHDL model does not need to be recompiled when the trigger or driver program changes. More detail on the trigger and driver components and how they are used to construct a mixed-level interface for a



```
trigger
10  wait_on_probe
20  case_probe_is
30     when 0000
40     -- this is the normal
50     -- operating condition
60       output 1 after 0
70     when 1000
80       -- this is the reset
condition
90       output 2 after 0
100 end_case
```

```
driver
10  wait_on_token
20  case_token_is
30     when 1
40        -- reset <= 1 after 5 ns
50         -- (normal operating
condition)
60        output 1 after 5
70     when 2
80        -- reset <= 0 after 5 ns
90        -- (reset condition)
100       output 0 after 5
110 end_case
```

Figure 19. The watch-and-react interface

microprocessor based system is available in [19].

Three papers which describe the mixed-level modeling improvements added to the ADEPT system are attached as appendices to this report. The CSIS technical report entitled “Hybrid Modeling with Synchronous Interpreted Elements” by Meyassed et. al., describes the mixed-level modeling methodology and interface for SDE components in more detail. The paper entitled “Mixed-level modeling in VHDL using the Watch-and-React Interface” by Dungan et. al., describes the mixed-level interface for complex sequential interpreted models in more detail. Finally, “A Top-down Design Environment for Developing Pipelined Datapaths” by McGraw et. al., describes a library of modeling elements for modeling pipelined datapaths added to ADEPT under this project. This library of elements was intended to aid in modeling cycle-based systems such a microprocessors, in ADEPT at an abstract RTL level. The ultimate goal of adding this library and modeling capability to ADEPT was to provide an area of application for the mixed-level modeling techniques using combinational interpreted components available in ADEPT which were described previously.

## 7. Task 4 - Integration and Tool Improvements

The goal of this task was to incorporate the technologies, tools, and libraries of modeling modules developed in the tasks described above, into the deliverable version of ADEPT. In addition, several improvements to the basic ADEPT framework were also developed under this task and included in the deliverable version of the tools. This includes a more robust and portable EDIF to VHDL translator which allowed a version of the ADEPT tools to be ported to a PC platform using OrCAD's Capture for Windows schematic capture tool and Model Technologies VSystem for Windows VHDL simulator.

In addition to the above, number of new post-simulation data visualization tools have been added to ADEPT. The first of these is AnimateADEPT, shown in Figure 20. AnimateADEPT allows the user to trace the status fields of tokens in the ADEPT model or individual tag fields and display the data back on the schematic interactively. In the case of the status field information, it is displayed on the schematic by changing the colors of the appropriate signals to reflect their values during the simulation. The user can then visualize the token passing protocol in the actual model as simulation time progresses. This is a very valuable tool for debugging ADEPT models. Similarly, the values of individual tag fields can be displayed on each signal with respect to simulation time.

Another post-simulation tool that has been added is the BAARS visualization tool shown in Figure 21. The BAARS tool allows the user to display the latency, utilization, throughput, or queue lengths in the ADEPT model dynamically as moving bar graphs as the simulation time advances. When the animation of the simulation data is finished, any of the performance metrics can be displayed on a line graph vs. simulated time as shown in the figure.

Finally, a visualization tool called Timeline was developed to display module utilization vs. simulation time as shown in Figure 22. Timeline displays a horizontal bar graph of module utilization where the times that a module is being utilized are shown as a filled bar and the times where a module is idle are empty. This allows the user to easily see excess idle time and concurrency in parallel processing applications.

There are a number of other documents, not all included herein, that outline the work

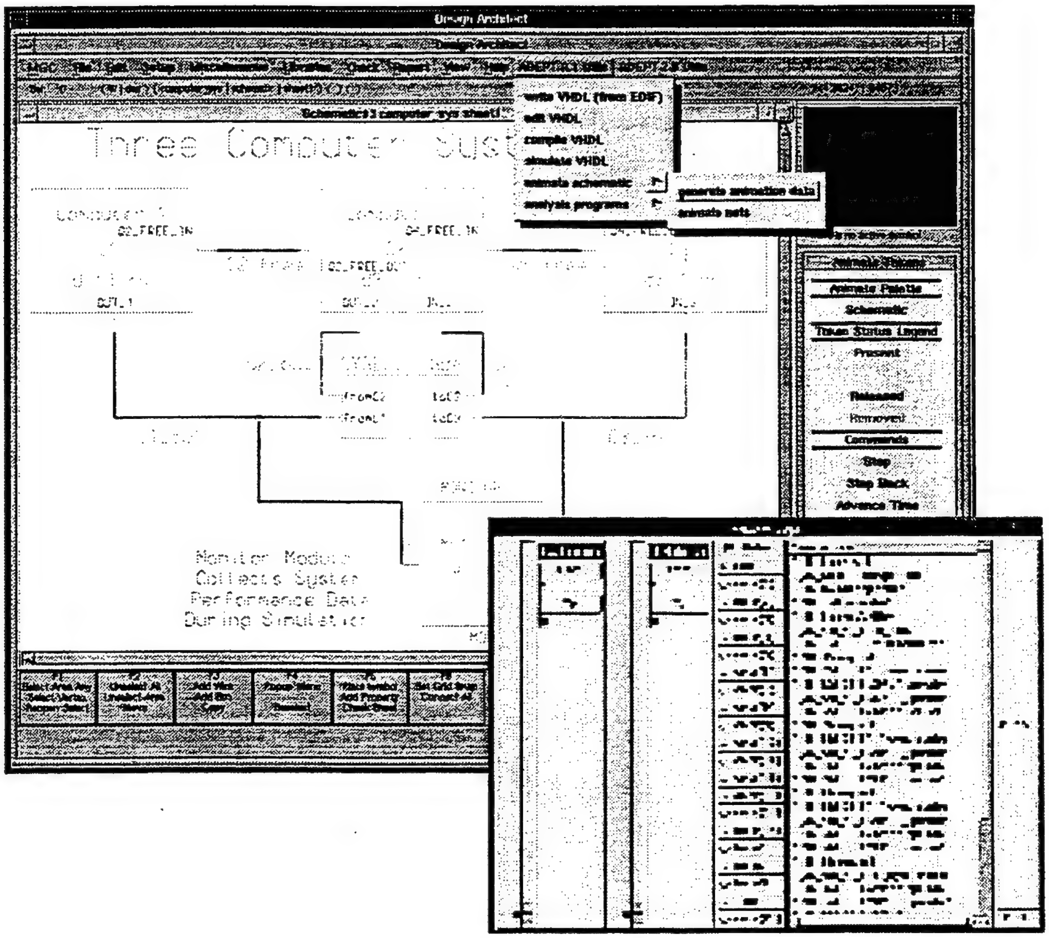


Figure 20. The AnimateADEPT tool

performed under this task. The documentation of the most recent deliverable version of ADEPT, version A.1, is available separately from this report. This includes the Unified Modeling Reference Manual which outlines the basic principals of ADEPT and how it is implemented in VHDL, the ADEPT A.1 Library Reference Manual, which includes data sheets on the over 150 modules in the various ADEPT modeling libraries, and the ADEPT Version A.1 Tutorial, Vantage and QuickVHDL versions, which describes how to use the ADEPT modules and tools to construct and analyze system-level models. Finally, the RASSP E&F educational module on Token-Based Performance Modeling using VHDL is included. This module was developed jointly under this project and the UVa portion of the RASSP E&F project and describes the goals and objectives of performance modeling, the background of performance modeling, in terms of

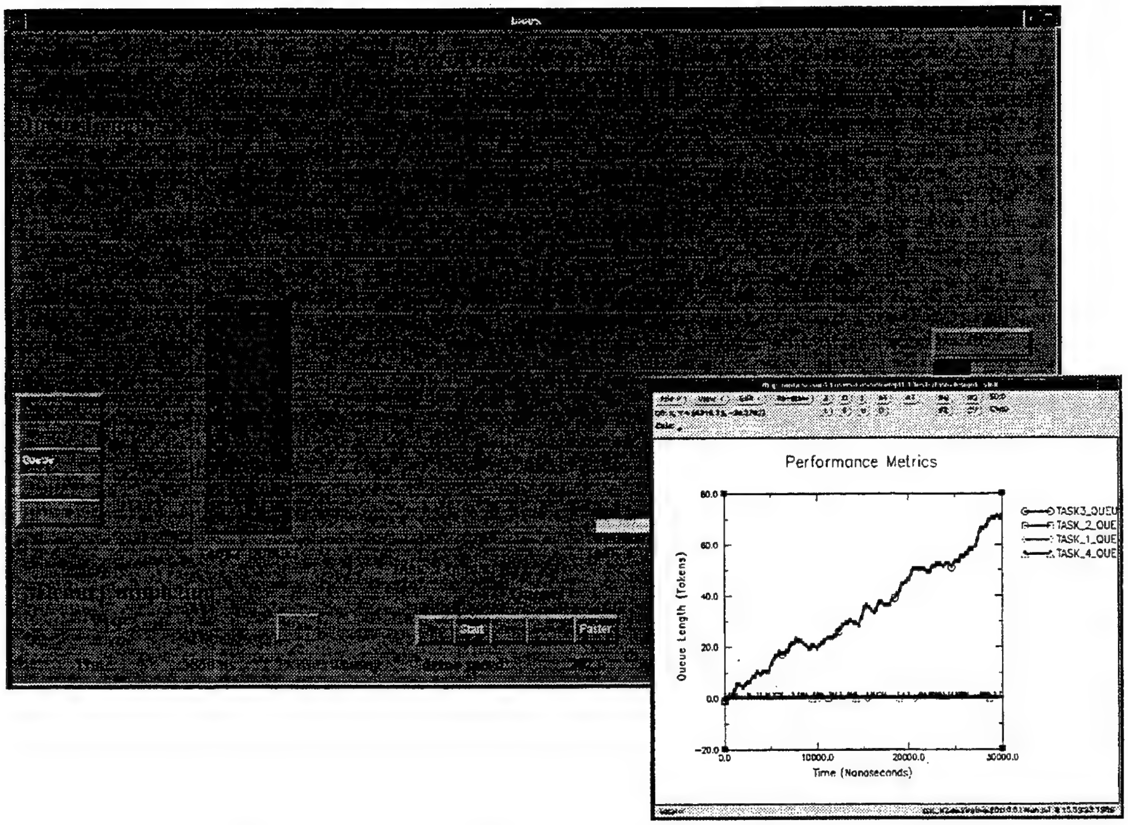


Figure 21. The BAARS performance visualization tool

traditional techniques like queuing networks, Petri Nets and non-VHDL based performance modeling tools, and VHDL-based performance modeling techniques and tools that are available, including ADEPT.

## 8. Conclusions

An integrated design environment called ADEPT that supports performance and dependability modeling at the system level has been developed. ADEPT supports the analysis of performance and dependability measures from the same high-level model and provides the capability to refine this high-level model into an implementation using mixed-level modeling. Significant additions have been made to the ADEPT environment under the RASSP program including mixed-level interfaces for sequential interpreted components, high-level dependability analysis and dependable systems codesign capabilities, and new application specific modeling libraries and post-simulation analysis tools.

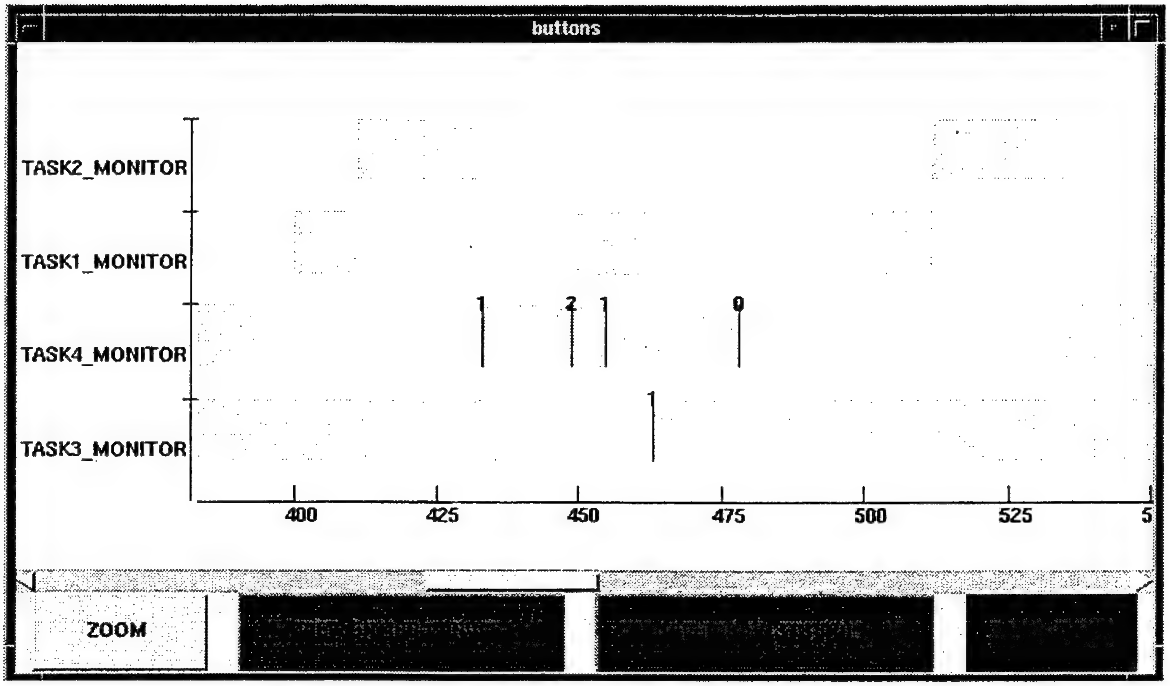


Figure 22. The Timeline performance visualization tool

## 9. Published Papers

This section lists the ADEPT related papers published by students, staff, and faculty supported by the RASSP Technology Base contract.

Swaminathan, G., Aylor, J. H., and Johnson, B. W., "Model Reduction Techniques Using Colored Petri Nets," *Proceedings of the 1993 TECHCON*, Atlanta, Georgia, October 1993, pp. 291-293.

Kumar, S., J. H. Aylor, B. W. Johnson, W. A. Wulf, "A Framework for Hardware/Software Codesign," *IEEE Computer*, vol. 26, December 1993, pp. 39-45.

Rao, R., G. Swaminathan, B. W. Johnson, J. H. Aylor, "Synthesis of Reliability Models from Behavioral Performance Models," *Proc. 1994 Annual Reliability and Maintainability Symposium*, Anaheim, California, Jan. 24 -27 1994, pp 292-297.

Swaminathan, G., R. Rao, J. H. Aylor, B. W. Johnson, "A VHDL based Environment for System Level Design and Analysis," *Proceedings of the VHDL International Users Forum*, San Francisco, May 1 - 4, 1994.

Kumar, S., J. H. Aylor, B. W. Johnson, W. A. Wulf, "Object-Oriented Techniques in Hardware Design," *IEEE Computer*, vol. 27, June 1994, pp. 64-70.

Kumar, S, R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, R. Waxman, "ADEPT: A Unified System Level Modeling Design Environment," *Proceedings of the 1st Annual RASSP Conference*, August 1994, pp. 114 - 123.

Klenke, R. H., ADEPT, "A System Level Modeling Environment Utilizing Mentor Graphics Tools," *1994 Mentor Graphics Users Group Conference*, October 1994.

RASSP Project Overview Version 1.0, CSIS Technical Report No. 940810.0, Department of Electrical Engineering, University of Virginia, August 1994.

ADEPT (Advanced Design Environment Prototype Tool) Version 1.1 Installation Manual, *CSIS Technical Report No. 941213.0*, Department of Electrical Engineering, University of Virginia, December, 1994.

ADEPT (Advanced Design Environment Prototype Tool) Tutorial, *CSIS Technical Report No. 941215.0*, Department of Electrical Engineering, University of Virginia, December, 1994.

Unified Modeling (UM) Reference Manual, *CSIS Technical Report No. 941216*, Department of Electrical Engineering, University of Virginia, December, 1994.

ADEPT Library Reference Manual, *CSIS Technical Report No. 950103.0*, Department of Electrical Engineering, University of Virginia, January, 1995.

Meyassed, M., J.H. Aylor, "Performance Modeling of DSP Elements Using Adept: The FIR Digital Filter," *CSIS Technical Report No. 950104.0*, Department of Electrical Engineering, University of Virginia, January, 1995.

Kumar, S., J. H. Aylor, B. W. Johnson, W. A. Wulf, R. D. Williams, "An Abstract Hardware/Software Model for Early Performance Evaluation," *Proceedings of the IEEE International Symposium on Systems Engineering of Computer Based Systems*, vol. 27, March 1995, pp. 140-145.

Rao, R., A. Rahman, B. W. Johnson, "Integrated Performance and Dependability modeling using the Advanced Design Environment Prototype Tool ADEPT." *Proceedings of the AIAA Conference on Computing in Aerospace 10*, San Antonio, Texas, March 28-30, 1995.

Meyassed, M., R.A. MacDonald, R.D. Williams, R.H. Klenke, and J.H. Aylor, "A Framework for the Implementation of Hybrid Modeling," *CSIS Technical Report No. 950421.0*, Department of Electrical Engineering, University of Virginia, April, 1995.

Rao, R., B. W. Johnson, R. D. Williams, J. H. Aylor, "Integrated Reliability and Performance Evaluation using Information Flow Models," *Proceedings of the 2st Annual RASSP Conference*, July 1995, pp. 109-114.

Meyassed, M., R. McGraw, J. H. Aylor, R. H. Klenke, R. D. Williams, F. Rose, J. Shackleton, "A Framework for the Development of Hybrid Models," *Proceedings of the 2st Annual RASSP Conference*, July 1995, pp. 147-154.

Voss, A. P., R. H. Klenke, J. H.Aylor, "The Analysis of Modeling Styles for System Level VHDL Simulations," *Proceedings of the VHDL International Users' Forum*, pp. 1.7-1.14, October 1995.

McGraw, R. M., M. Meyassed, R. H. Klenke, J. H. Aylor, R. D. Williams, "Refinement of System-Level Designs Using Hybrid Modeling," *International Conference on Engineering of Complex Computer Systems*, November 6-10, 1995.

Dungan, W. W., R. H. Klenke, James H. Aylor, "Heterogeneous Modeling with ADEPT and PML: Using the PML Lightweight Processor in the Lockheed Martin ATL SAR Model," *CSIS Technical Report No. 951218.0*, December 18, 1995.

Voss, A. P., R. H. Klenke, J. H. Aylor, "Simulation Time Reduction as Applied to the Martin ATL SAR Model," *CSIS Technical Report No. 951219.0*, 19 December, 1995.

Rao, R., A. Rahman, B. W. Johnson, "Reliability Analysis Using the ADEPT-REST Interface," *Proc. 1996 Annual Reliability and Maintainability Symposium*, January 22-25 1996, Las Vegas, Nevada.

Kumar, S., Aylor, J. H., Johnson, B. W., Wulf, W. A., and Williams, R. D., "Early Performance Evaluation Using Abstract Hardware/Software Models," *Proceedings of the VHDL International Users' Forum*, Spring Conference, Santa Clara, California, February 27-March 2, 1996, pp. 51-60.

Dungan, W. W., R.H. Klenke, and J.H. Aylor, "A 'Watch-and-React' Interface for Hybrid Modeling," *CSIS Technical Report No. 960531.0*, Department of Electrical Engineering, University of Virginia, May 1996.

ADEPT A.1 Library Reference Manual, *CSIS Technical Report No. 960625.0*, Department of Electrical Engineering, University of Virginia, June, 1996.

Unified Modeling (UM) Reference Manual (ADEPT A.1 Version), *CSIS Technical Report No. 960620.0*, Department of Electrical Engineering, University of Virginia, June, 1996.

Meyassed, M., R. M. McGraw, R. H. Klenke, J. H. Aylor, B. W. Johnson, "ADEPT: A Unified Environment for System Design," *The RASSP Digest*, Vol. 3, November 1996, pp. 18-21.

Kumar, S. R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, R. Waxman, "ADEPT: A Unified Environment for End-to-End System Design," *Journal of Current Issues in Electronic Modeling*, Vol. 4, Kluwer Academic Publishers, 1996, pp. 55-82.

Kumar, S., Aylor, J. H., Johnson, B. W., and Wulf, W. A., "Object-Oriented Modeling of Hardware/Software for Embedded Systems", *Journal on Current Issues in Electronic Modeling*, Vol. 7, Kluwer Academic Publishers, 1996.

Kumar, S., Aylor, J. H., Johnson, B. W., Wulf, W. A., and Williams, R. D., "A Model for Exploring Hardware/Software Trade-Offs and Evaluating Design Alternatives," *Journal on Current Issues in Electronic Modeling*, Vol. 8, Kluwer Academic Publishers, 1996.

Choi, C. Y., Johnson, B. W., and Bechta-Dugan, J., "Dependable System Co-Design Using Data Flow Models," *Proceedings of the 1997 IEEE Annual Reliability and Maintainability Symposium*, Philadelphia, Pennsylvania, January 20-23, 1997, pp. 263-270.

Klenke, R. H., A. P. Voss, J. H. Aylor, "Performance Modeling of Multicomputer Systems in VHDL using ADEPT," *Proceedings of the IASTED International Conference on Modeling and Simulation*, May 1997, pp. 429-438.

Klenke, R. H., M. Meyassed, J. H. Aylor, B. W. Johnson, R. Rao, A. Ghosh, "An Integrated Design Environment for Performance and Dependability Analysis," *Proceedings of the ACM Design Automation Conference*, June 1997 pp. 184-189.

Choi, C. Y., Johnson, B. W., and Profeta, III, J. A., "Safety Issues in the Comparative Analysis of Dependable Architectures," *IEEE Transactions on Reliability*, to appear.

Dungan, W. W., R. H. Klenke, J. H. Aylor, "Mixed-Level Modeling in VHDL Using the Watch-and-React Interface," *Proceedings of the VHDL International Users Forum*, Fall 1997, pp. 25-32.

Klenke, R. H., J. H. Aylor, B. W. Johnson, M. Meyassed, W. W. Dungan, C. Y. Choi, R. Rao, "Improvements to ADEPT - A VHDL Based Integrated Design Environment for Performance and Dependability Analysis," *Proceedings of the VHDL International Users Forum*, Fall 1997, pp. 190-199.

Choi, C. Y., B.W. Johnson, J. Bechta Dugan, "Dependable System Codesign Using Data Flow Models," *Proceedings of the Reliability, Availability, and Maintainability Symposium 1997.*

Mcgraw, R. M., R. H. Klenke, J. H. Aylor, "A Top-Down Design Environment for Developing Pipelined Datapaths," *Proceedings of the ACM Design Automation Conference*, June 1998 (to appear).

Choi, C. Y., B.W. Johnson, J. Bechta Dugan, "Data Flow Codesign Models for Dependable Systems", *Proceedings of the 27th Fault Tolerant Computing Symposium*, (submitted).

Choi, C. Y., B.W. Johnson, J. Bechta Dugan, "Assurance of Hardware/Software Codesign Models Using Operator Nets," *COMPASS '97*, (submitted).

## 10. References

[1] ASIC & EDA, January 1993

[2] IEEE, *"IEEE Standard VHDL Language Reference Manual,"* New York, NY, IEEE Std. 1076-1993, June 6, 1994.

[3] Kumar, S, R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, R. Waxman, "ADEPT: A Unified System Level Modeling Design Environment," *Proceedings of the 1st Annual RASSP Conference*, August 1994, pp. 114 - 123.

[4] Jensen, K., "Colored Petri Nets: A high level language for system design and analysis," in *High-level Petri Nets: Theory and application,* K. Jensen and G. Rozenberg (Eds.), Berlin: Springer-Verlag, 1991, pp. 44-119.

[5] Hady, F. T., *A Methodology for the Uninterpreted Modeling of Digital Systems in VHDL,* Master of Science Thesis, Dept. of Electrical Engineering, University of Virginia, January 1989.

[6] Klenke, R. H., A. P. Voss, J. H. Aylor, "Performance Modeling of Multicomputer Systems in VHDL using ADEPT," *Proceedings of the IASTED International Conference on Modeling and Simulation*, May 1997, pp. 429-438.

[7] Swaminathan, G., R. Rao, J. H. Aylor, B. W. Johnson. "Colored Petri Net Descriptions for the UVA Primitive Modules," University of Virginia, CSIS Technical Report No. 920922.0, September 22, 1992.

[8] Dennis, J. B., "Modular, Asynchronous Control Structure for a High Performance Processor," *ACM Conference Record, Project MAC,* Massachusetts, 1970, pp. 55-80.

[9] *ADEPT A.1 Library Reference Manual*, CSIS Technical Report 960625.0, University of Virginia, June, 1996.

[10] Voss, A. P., R. H. Klenke, J. H. Aylor, "The Analysis of Modeling Styles for System Level VHDL Simulations," *Proceedings of the VHDL International Users Forum*, Fall 1995, pp. 1.7-1.13.

[11] Voss, A. P., *Analysis and Enhancements of the ADEPT Environment*, Master of Science Thesis, Dept. of Electrical Engineering, University of Virginia, May 1996.

[12] Rao, R., G. Swaminathan, B. W. Johnson, and J. H. Aylor, "Synthesis of Reliability Models from Behavioral-Performance Models," Proceedings of the 1994 Reliability and Maintainability, Symposium (RAMS), January 1994, pp. 292-297.

[13] Molloy, M. K., *Performance Analysis Using Stochastic Petri Nets*, IEEE Transactions on Computers, Vol. 31, No. 9, Sept. 1982, pp. 913-917.

[14] Cutright, E. D., and B. W. Johnson, "A Simulation-based Approach to Integrated Performance and Reliability Modeling using VHDL," Proceedings of the 1994 Reliability and Maintainability Symposium (RAMS), January 1994, pp. 402-408.

[15] Kumar, S., J. H. Aylor, B. W. Johnson, W. A. Wulf, *The Codesign of Embedded Systems: A Unified Hardware/Software Representation*, Kluwer Academic Publishers, 1996.

[16] Meyassed, M., R. M. McGraw, J. H. Aylor, R. H. Klenke, R. D. Williams, F. Rose, and J. Shackleton, "A Framework for the Development of Hybrid Models," Proceedings of the 2nd Annual RASSP Conference, July, 1995, pp. 147-154.

[17] MacDonald R. M., *Hybrid Modeling of Systems with Interpreted Combinational Elements*, Ph.D. Dissertation, Dept. of Electrical Engineering, University of Virginia, May 1995.

[18] Meyassed M., *System Level Design: Hybrid Modeling with Sequential Interpreted Elements*, Ph.D. Dissertation, Dept. of Electrical Engineering, University of Virginia, January 1997.

[19] Dungan, W. W., *ADEPT Simulation and Modeling Enhancements,* Master of Science Thesis, Dept. of Electrical Engineering, University of Virginia, May 1997.

# APPENDIX A - Relevant Papers & Technical Reports

# A VHDL Based Environment for System Level Design and Analysis

Gnanasekaran Swaminathan, Ramesh Rao, James H. Aylor, and Barry W. Johnson
University of Virginia, Charlottesville

## Abstract

*The UVA uninterpreted modeling methodology uses a set of predefined primitive elements to model computer systems that can be used to explore different design alternatives. In this paper, we present an overview of the VHDL perspective of our UVA design methodology. We present Colored Petri Net models for the primitive elements that formalizes the UVA methodology. We then present a translation algorithm which translates the Petri Net (PN) model to VHDL so that the PN model can be simulated. In order to speed up the simulation, we also present a set of reduction rules that reduces the complexity of the PN model.*

## I Introduction

As the set of possible designs satisfying a given specification at the system level is very large, it is difficult to pick a good design that has the maximum performance for a given cost. The UVA uninterpreted modeling methodology allows a designer to identify performance bottlenecks and to quickly find the effect of any design decisions on the performance of the system [1]. Thus, by eliminating the designs that give poor performance, the UVA methodology reduces the design space. Also, the UVA methodology uses simple primitive modules to model computer systems thereby eliminating the need for the designer to understand complex queuing theory or Petri Net (PN) theory to analyze the performance of the system. That is, the UVA methodology enhances the productivity of the designer by reducing the time and the cost needed to arrive at a good design.

A problem facing fault-tolerant designers is the high level of modeling expertise required by the current reliability tools to produce meaningful results. This high level of expertise has often had the effect of postponing reliability calculations until near the end of the design process, and has often required the services of a reliability expert, rather than the system designer, to perform the dependability analysis [2]. The UVA methodology, by providing facilities for automated reliability analysis of a model constructed using the primitive modules, eases the design of fault-tolerant systems to a great extent.

Besides integrating performance and dependability evaluation tools into a Computer Aided Design environment, UVA methodology enables a design engineer to capture the following information about a system in a single unified model:

1. Data-flow and control-flow through the system
2. Input-Output relationships of components and sub-systems
3. Data processing rates and component delay information
4. Dependability characteristics of the components such as failure rates and coverage

By using a single model, our methodology eliminates the inconsistency between different models used to perform system level analysis and trade-offs. The primitive modules may be interconnected to model hardware, software, and the interaction between the two. The designer can independently refine elements of the design and simulate the system models in which different components are described at different levels of abstraction and interpretation [1].

The behavior of each primitive module is defined by a Colored Petri Net (CPN) [3] which provides an unambiguous mathematical specification of the module and a VHDL description which has a one-to-one correspondence with the CPN defined behavior of the module. Thus, a model built from the primitive modules can be simulated using a VHDL simulator and at the same time analyzed using CPN theory by converting the model into its corresponding CPN representation. The system model can be simplified by reducing the CPN model using a set of CPN reduction rules [4]. The reduced CPN model can be simulated in the same VHDL environment if we can convert the reduced CPN model to VHDL. By simulating all our models in the same environment, we eliminate the errors due to changes in the environment and thus, get consistent simulation results. Also, the reduced CPN model speeds up the simulation.

In this paper, we concentrate on how we use VHDL in our methodology. First, we give an overview of the UVA design environment in Section II. In Section III, we briefly describe the CPN representations for the primitive modules. In Section IV, we present the CPN to VHDL translation algorithm. In Section V, we present the CPN reduction rules which reduce the CPN model. In Section VI, we illustrate how the same performance model can be

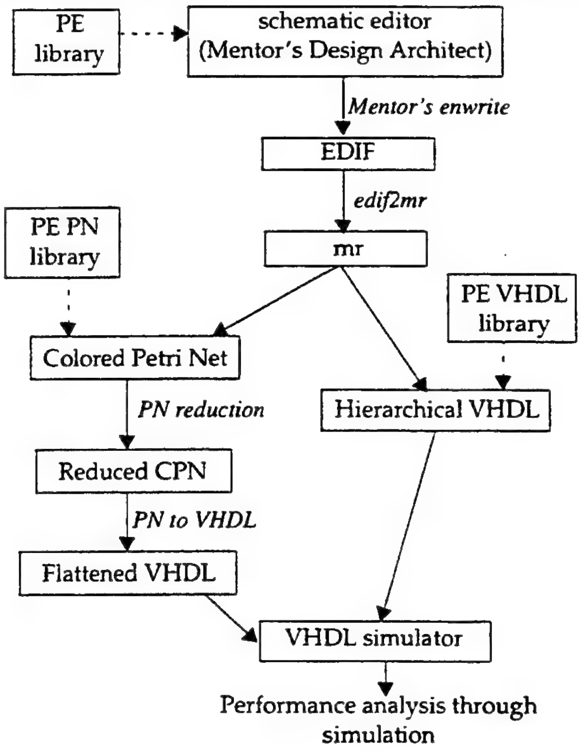
**Appeared in the Proceedings of the Spring VHDL International Users Forum, 1994, pp. 110-116**

used to study other aspects of the system like reliability. Finally, we present some experimental results in Section VII.

## II UVA design environment

The UVA environment allows the user to capture the primitive element model of the system using Mentor Graphics' Design Architect schematic editor. Once the primitive element model is created using the schematic editor, the user can study the performance of the system through simulation. We can simulate the hierarchical VHDL obtained by directly substituting each primitive element by its corresponding VHDL or we can reduce the equivalent CPN and then, translate the reduced CPN to VHDL and simulate the resulting VHDL. We use CPN reduction in the CPN to VHDL path in Figure 1 to simplify the PN model in the hope of reducing the simulation time.

In order to accommodate both VHDL and Petri Nets (PN), we use an internal representation of the primitive modules called *mr*. A UVA module can be represented in VHDL, in Petri Nets, or as a netlist of other modules in mr. No other format lets the mixing of Petri Nets and VHDL as *mr* lets us to do. Once a primitive element model of a system is translated into the internal *mr* representation, it can be further translated into VHDL through two different paths as described before.



PE = primitive element

**Figure 1: UVA design environment.**

## III Colored Petri Net models

The UVA modules are divided into three categories—control, color, and delay. The control modules are used to control the flow of data. The color modules are used to modify the state information. The delay modules help simulate the time delays in the system. The fault modules are special color modules that are used to study the fault tolerant characteristics of the system.

We use Jensen's Colored PNs (CPN) to model the modules. The CPN models are more succinct than other types of PNs like ordinary PNs and stochastic PNs [3]. They achieve their succinctness by distributing their complexity into a) net structure, b) descriptions, and c) net inscriptions.

The CPN structure is a bipartite directed graph. Figure 2 shows the CPN model of an example control module. The places (circles), the transitions (bars), and the arcs connecting them form the structure of the CPN. The CPN places can contain tokens. Unlike other PN tokens, the CPN tokens can have complex data types called colors. The data types of the tokens and the variables used in the net inscriptions are described in CPN descriptions (see the declaration of x and y variables in Figure 2 for an example).

The CPN inscriptions called arc expressions written by the side of an input (output) arc indicate what tokens to remove (add) when the transition associated with the arc fires. The inscriptions written by the side of a transition (called guard) describe the additional conditions that need to be satisfied for the transition to fire. Normally, a transition will fire if all the input places have the tokens indicated by the corresponding arc inscription.

The switch module, shown in Figure 2, sends the ready token arriving at input i1 to output o1 if a control token is present at the control input ci1. The control token in place ci1 is not removed as n is 0 in n'y. If n is 1, it is omitted as in x. The switch module then waits for an acknowledge token to arrive on the output o1. When the acknowledge

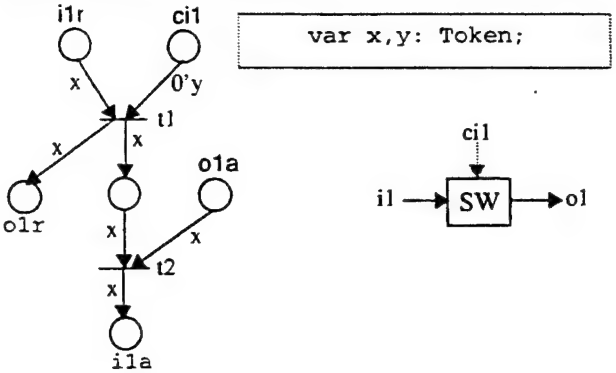


**Figure 2: CPN model of the switch control module.**

**Appeared in the Proceedings of the Spring VHDL International Users Forum, 1994, pp. 110-116**

token arrives on the output o1, the switch module acknowledges its input i1.

For a detailed description of each of the modules and the CPN model for each, the reader is referred to [5].

## IV Petri Net to VHDL translation algorithm

In this section, we present an algorithm to translate a PN into VHDL. We also analyze the space and time complexity of the algorithm.

The arc expressions of the PN will evaluate to the enum constant a) *minus* if the arc is (place, trans) and the arc inscription is n'y and n > 0 and the place has at least n tokens, b) *zero* if the arc is (place, trans) and the arc inscription is 0'y and the place has at least 1 token, c) *not* if the arc is (place, trans) and the arc inscription is -1'y and the place has no tokens in it, d) *plus* if the (trans, place) arc's inscription is n'y, e) *repl* if the (trans, place) arc's inscription is =n'y, and f) *nop* otherwise.

*Simulation rule:* A transition is enabled if none of its input arc expressions evaluate to *nop* An enabled transition fires by removing n tokens (if the arc inscription is n'y) from the places whose arc expressions evaluate to *minus*, adding n tokens (if the arc inscription is n'y) to the places whose arc expressions evaluate to *plus*, and replacing the tokens in the places whose arc expressions evaluate to *repl* with n tokens (if the arc inscription is =n'y).

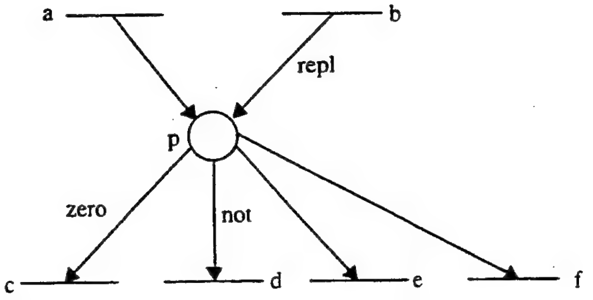
From the simulation rule, it is clear that we consider at most one color type per place. Also, if we use the following color for the PN token, we need to use only one color type for all the places in the PN:

```
type pntoken is record
    num: integer;
    tk: token; -- uva primitive module token
end record;
```

Since, each data link and each control link in a network of primitive modules hold at most one UVA primitive module token, the PN models of the primitive modules can be built with at most one token per place. We also assume the same in the algorithm.

A place activates a VHDL signal of type `pntoken` whenever it gets or gives away its token. A transition also activates a VHDL signal of type `pntoken` whenever it fires. We make each place and each transition a separate process. Each process will update only its corresponding signal.

A place gets a new token when one of its input transitions with *plus* or *repl* arc expression fires, and it gives away its token when one of its output transitions with *minus* arc expression fires. So, a place process depends on all its input transitions, and all the output transitions with *minus* arc expression. *Example:* The place process for the place p in the PN of Figure 3a is given in Figure 3b. The process p depends on the input transitions a and b and output transitions e and f. Since, output transition arcs for c and d are not *minus*, the process p does not depend on them. If one of the input transition signals is active, then the process p updates the pntoken signal p with the active transition signal. If any one of the output transition signals whose arcs evaluate to *minus* is active, then the process p resets the token count of the `pntoken` signal p.



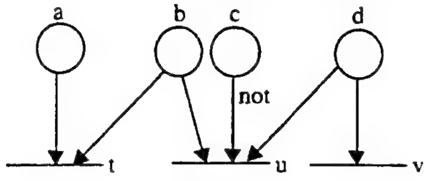
(a)

```
p: process (a, b, e, f)
begin
   if a'active then
      p <= (p.num + 1, a.tk);
   elsif b'active then
      p <= (1, b.tk);
   elsif e'active
          or f'active
      p.num <= p.num - 1;
   end if;
end p;
```

(b)

**Figure 3: Place process example.**

A transition is enabled when all its input places with *minus* or *zero* arc expressions have tokens and all input places with *not* arc expressions have no tokens. An enabled transition may or may not fire. For example, consider the PN shown in Figure 4a wherein if the places with *minus* arc expressions a, b, and d have tokens and the place with the *not* arc expression c does not have any token, then transitions t, u, and v will all be enabled. But, because of conflict only two of them can fire. We assign a priority to the firing of the transitions to resolve conflicts. In Figure 4a, we have assigned transition t the highest priority followed by u and v. So, the transition process u has to check whether transition signal t is active or not before it activates signal u. In a similar manner, process v will check whether transition signal u is active before it activates signal v. Whereas, process t does not have to check any transition process as it has the highest priority to fire.



(a)

```
u: process (t, b, c, d)
begin
   if b.num = 1 and c.num =0
      and d.num = 1 then
      -- trans u is enabled
      if not t'active then
         -- conflict resolved
         u <= b;
      end if;
   end if
end u;
```

(b)

**Figure 4: Transition process example.**

```
for each node n in the PN
   declare a pntoken signal as follows:
   signal n: pntoken;
   if (n is a place) then
      generate_place_process (n);
   else
      generate_trans_process (n);
   end if;
end for;
```

**Figure 5a: PN to VHDL algorithm—main function.**

The main function that translates our PN to VHDL is presented in Figure 5a. It declares a signal of type `pntoken` for each PN node and then, if the node is a place, it generates a place process by calling `generate_place_process()` or if the node is a trans, it generates a transition process by calling `generate_trans_process ()`.

The `generate_place_process ()` is presented in Figure 6b. The generated VHDL code checks whether each node in the input node list is active. If a node is active and the arc expression is *plus*, then it increments the token count of the place's `pntoken` by 1 and copies the UVA token of the input node to the place's `pntoken`. After that, if any of the output node with the *minus* arc expression is active, it decrements the token count of the place's `pntoken` by 1.

The `generate_trans_process ()` function is presented in Figure 7c. It generates a VHDL process which first checks whether the transition is enabled or not. Once the transition is enabled and none of the transitions involved in the conflict that has higher firing priority than the current transition node has fired, the generated VHDL process copies the `pntoken` of the first input place with the *minus* arc expression in the input place list to the transition signal.

Since the translation algorithm generates one signal and one process for each node, it takes $O(n)$ time. If the degree of a node (the total number of input and output nodes connected to the node) is a constant, then the algorithm takes $O(n)$ space.

## V Petri Net reduction rules

The PN reduction rules presented here are given in terms of ordinary PN. They are used to reduce the PN for the control modules part of the complete model. The PNs for the control modules do not manipulate color and hence, they can be treated as ordinary PNs.

Figure 8 illustrates the application of one of our rules.

```
function generate_place_process
                                       (node p)
   ipl = input place list;
   for n in output trans list do
      if (p, n) is minus
         opl->append (n);
      end if;
   done;
   generate:
   p: process (ipl, opl)
   begin
      for a in ipl loop
         if a'active then
            if (a, p) is plus then
               p <= (p.num + 1, a.tk);
            else
               p <= (1, a.tk);
            end if;
            exit loop;
      end for;
      if any a in opl is active then
         p.num <= p.num - 1;
      end if;
   end p;
end function
```

**Figure 6b: PN to VHDL algorithm—place process generator.**

**Appeared in the Proceedings of the Spring VHDL International Users Forum, 1994, pp. 110-116**

```
function generate_trans_process
                               (node t)
   tl  = transitions involved in conflict with t
   htl  = a ∈ tl  and  priority of a higher than t
   impl  =  list of input places with minus arc
            expression
   inpl  =  list of input places with not arc
            expression
   generate:
   t: process (htl, impl, inpl)
      if ∀a ∈ impl check a.num = 1
         and ∀a ∈ inpl check
               a.num = 0 then
         if ∀a ∈ htl check
               a'active = false then
            t <= head(impl);
         end if;
      end if;
end function;
```

Figure 7c: PN to VHDL—transition process generator.

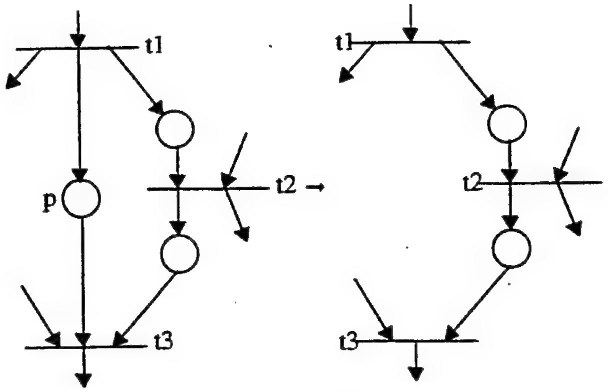


Figure 8: Application of a reduction rule.

The rule that is applied states that if transition $t_c$ can fire only after transitions $t_a$ and $t_b$ fire, and transition $t_b$ can fire only after transition $t_a$ fires, then remove any places that exclusively connect transitions $t_a$ and $t_c$.

In addition to our rules, we also use the rules given in [6]. We describe all the rules in great detail in [4].

## VI Reliability analysis using the primitive modules

A subset of the primitive modules referred to as the *Fault* modules are used to model several dependability characteristics of a system including failure, fault detection, error correction, and reconfiguration strategies. We have a set of abstraction rules which can be effectively used to extract the analytical reliability model from the primitive element model of a system [8]. For example, Figure 9 illustrates the application of the abstraction rules to the voter of a Triple Modular Redundancy (TMR) system. The abstraction rules simply throw away all the information that are not needed for the reliability analysis. Figure 10 illustrates how the abstracted TMR CPN model is converted into traditional Markov model for reliability analysis. The reader is referred to [8] for more information on the reliability analysis in the UVA methodology.

## VII Experimental results

PN models for a three computers system that share a common bus and the ATAMM multiprocessor system described in [7] are built and reduced using the application of our rules. We have obtained significant reductions as is evident from Table 1.

Table 1: Savings in terms of PN nodes.

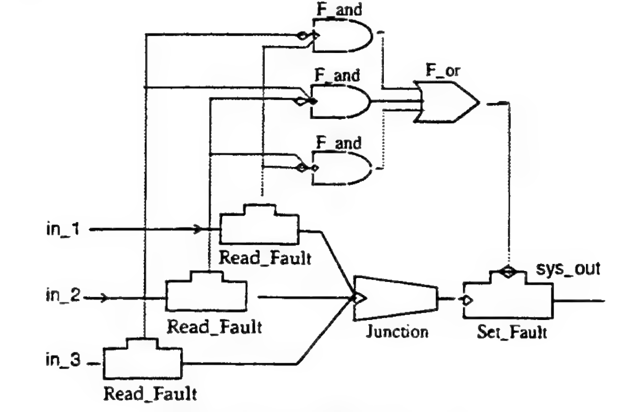
| Example | Nodes before reduction | Nodes after reduction |
|---|---|---|
| 3-computers | 166 | 68 |
| ATAMM | 735 | 360 |

Even though we have implemented the PN to VHDL translation algorithm, we have not merged the implementation into our main tool illustrated in Figure 1. So, we manually translated the PN for one of the computers in the 3-computers example into VHDL using the translation algorithm. The time to simulate both the 3-computers UVA model in which all the computers are modeled using the UVA modules and the 3-computers model in which one of the three computers is a PN model is recorded in Table 2. As can be seen in Table 2, the savings are 17% in cpu time and 24% in real time (wall clock time).

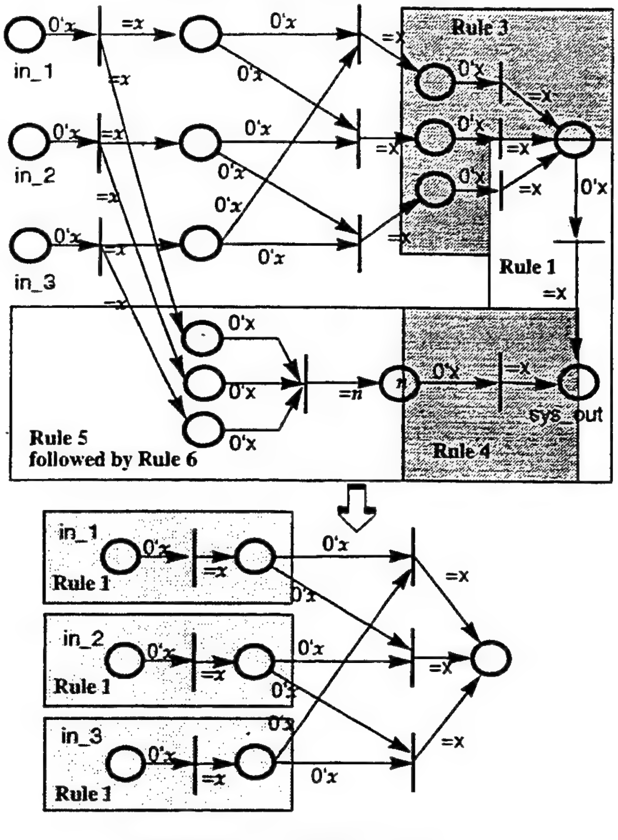
Table 2: 3-computers example simulation results.

|  | UVA model (sec) | PN model (sec) |
|---|---|---|
| cpu time | 17.8 | 6 |
| real time | 20.9 | 15.8 |

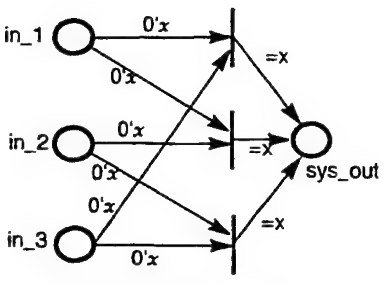
**Appeared in the Proceedings of the Spring VHDL International Users Forum, 1994, pp. 110-116**



(a) Voter Primitive Element Model

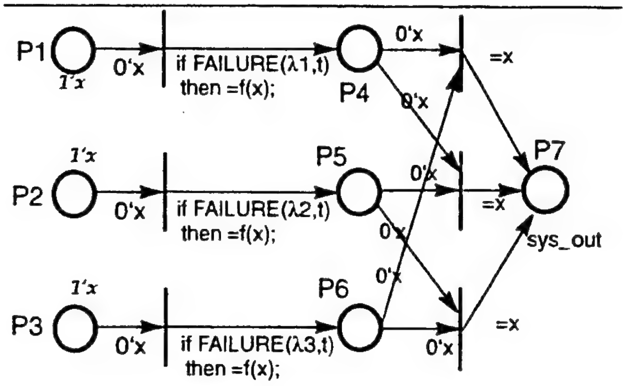


(b) Snapshots in the Reduction Process



(c) Reduced CPN Model

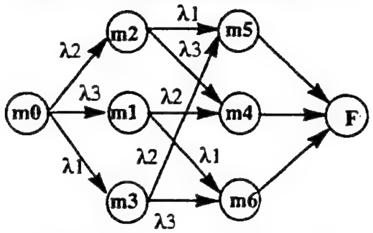
**Figure 9: Reduction of the Voter module**



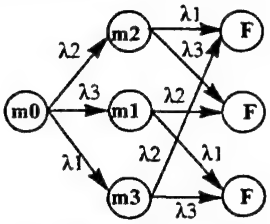
(a) Reduced System CPN Model

| | P1 | P2 | P3 | P4 | P5 | P6 | P7 |
|---|---|---|---|---|---|---|---|
| m0 | x | x | x | | | | |
| m1 | x | x | x | x | | | |
| m2 | x | x | x | | x | | |
| m3 | x | x | x | | | x | |
| m4 | x | x | x | x | x | | |
| m5 | x | x | x | x | | x | |
| m6 | x | x | x | | x | x | |
| m7 | x | x | x | x | x | x | |
| m8 | x | x | x | x | x | | x |
| m9 | x | x | x | x | | x | x |
| m10 | x | x | x | | x | x | x |
| m11 | x | x | x | x | x | x | x |

(b) Possible markings of the system CPN



(c) Corresponding Markov model



(d) Reduced Markov model

**Figure 10: Obtaining the Markov Model**

**Appeared in the Proceedings of the Spring VHDL International Users Forum, 1994, pp. 110-116**

## VIII Conclusion

In this short paper, we briefly described the VHDL based system level design environment. We used a set of predefined primitive elements, each of which has both an underlying Petri Net representation and a VHDL representation, to model a system. The PN models not only formalizes the UVA methodology but also lend themselves to transformations like PN reductions so that the complexity of the system level model is greatly reduced. We also presented an algorithm to translate the PN model to VHDL so that the PN model can be simulated. Finally, we gave a couple of examples which showed the reduction rules does indeed greatly simplify the original model.

## IX References

[1] S. S. Mitra, *"An interpretation interface for hybrid performance modeling,"* Charlottesville, VA: M.S. Thesis, Dept. of Elec. Eng., Univ. of Virginia, Jan. 1992.

[2] R. K. Iyer and W. H. Bryant, "Computer-Aided Design of Dependable Mission Critical Systems," *Proceedings of the 19th Fault-Tolerant Computing Symposium,* 1989, pp. 416-420.

[3] K. Jensen, "Colored Petri Nets: A high level language for system design and analysis," in K. Jensen and G. Rozenberg (Eds.), *"High-level Petri Nets: Theory and application,"* Berlin: Springer-Verlag, 1991, pp. 44-119.

[4] G. Swaminathan, J. H. Aylor, and B. W. Johnson, *"An $O(n^2 \log n)$ time Petri Net reduction algorithm,"* unpublished.

[5] G. Swaminathan, R. Rao, J. H. Aylor, and B. W. Johnson, *"Colored Petri Net descriptions for the UVA primitive modules,"* Charlottesville, VA: TR-920922.0, CSIS Lab., Univ. of Virginia, Sep. 1992.

[6] H. Lee-Kwang, J. Favrel, and P. Baptiste, "Generalized Petri Net Reduction Method," *IEEE Transactions on Systems, Man, and Cybernetics,* Vol. SMC-17, No. 2, Mar./Apr. 1987, pp. 297-303.

[7] E. D. Cutright, R. Rao, B. W. Johnson, and J. H. Aylor, *"Modeling an ATAMM Based Multiprocessor System using VHDL,"* Charlottesville, VA: TR-910111.0, CSIS, Dept. of Elec. Eng., Univ. of Virginia, Jan. 1991

[8] R. Rao, G. Swaminathan, B. W. Johnson, and J. H. Aylor, "Synthesis of Reliability Models from Behavioral Performance Models," *Proceedings of the 1994 Reliability and Maintainability Symposium,* Anaheim, CA, Jan. 1994, pp. 292-298.

**Appeared in the Proceedings of the Spring VHDL International Users Forum, 1994, pp. 110-116**

46

# The Analysis of Modeling Styles for System Level VHDL Simulations[1]

Andrew P. Voss, Robert H. Klenke, James H. Aylor,
Department of Electrical Engineering
University of Virginia
Charlottesville, VA 22903

## Abstract

*This paper presents the results of a study to examine the effects of various VHDL model characteristics on simulation execution times. Four different modeling characteristics of complex VHDL models were examined: the size of signals, the use of file input and output (I/O) operations, the use of bus resolution functions, and the overall size and complexity of VHDL models. To develop models and tests for these characteristics, the University of Virginia's Advanced Design Environment Prototyping Tool (ADEPT) was used. This performance modeling environment provided an easy framework to develop tests for the four characteristics to be examined.*

*After developing the different tests to examine these characteristics, multiple runs were conducted to minimize random variations due to processor loading. The results of these tests are presented here along with detailed explanations of how each test was developed and conducted. From the results presented here future VHDL model builders will be able to develop more efficient models by knowing the effects different model characteristics will have on their simulation execution times.*

## 1. Introduction

As the use of the VHSIC Hardware Description Language (VHDL) to describe complex systems grows, the simulation execution speed of VHDL becomes increasingly important. Quick execution of simulations for various modeling alternatives is required for efficient exploration of the design space. Further, as VHDL is used to describe systems at higher levels of abstraction, the use of large complex data structures and bus resolution functions is required. The use of these complex constructs can be at odds with the requirement for efficient simulation execution. This report presents the results of an investigation of the effect of several constructs typically used in high-level VHDL models on simulation execution times.

## 2. Background

In order to test the effect of various model characteristics on simulator execution time of VHDL models, the University of Virginia's ADEPT (Advanced Design Environment Prototype Tool) [1] environment was used. This performance modeling environment is based upon a building block approach where the basic building blocks are referred to as ADEPT modules. These ADEPT modules can be interconnected to create complex structures which represent systems. The individual behavior of these modules has been described in VHDL. The modules use a token passing mechanism to transfer information between modules. These tokens are composed of an array of integers that can be broken down into two different groups. The first group is the status field; this field is used to control the flow of the token. The status field can assume one of four values: present, acknowledged, released, or removed, and is used to implement a fully interlocked handshaking protocol. This status field is not modifiable by the user. The second group is composed of eighteen integer fields. All of these fields can be edited by the user and contain user specified data. These eighteen integer fields will be referred to as the tag fields. The use of this token structure means that all signals in an ADEPT simulation are composed of an array of nineteen integers.

## 3. Simulation Tests

There are four primary characteristics of VHDL models that will be examined in this paper: the size of signals, the use of file input and output (I/O) operations, the use of bus resolution functions, and the overall size and complexity of the model. Numerous tests have been implemented to evaluate the effect of each characteristic on VHDL simulation execution times.

---

[1] This work was sponsored by the Advanced Research Projects Agency under Contract No. F33615-93-C1313.

**Appeared in the Proceedings of the Fall VHDL International Users Forum, 1995, pp. 1.7-1.13**

### 3.1 Reduction of Signal Size

A set of tests has been developed to examine the effect of signal size, that is the number of fields in a token, versus the overall simulation execution time. The ADEPT system, since it incorporates an array of nineteen integers as a token, provides a convenient way to reduce the size of a signal throughout a VHDL model. Many models built using the ADEPT environment do not make full use of all eighteen of the tag fields. This situation allows for the size of a signal within a VHDL simulation to be reduced by simply modifying the size of the ADEPT token. By reducing the number of tag fields within the standard ADEPT token, the effect of signal size versus simulation execution time has been studied.

### 3.2 File I/O

During the examination of two different simulators, it was observed that these simulators handled file input and output differently. This result led to the investigation of the effect of file operations on VHDL simulation execution time. By comparing the simulation execution time for a system using VHDL models which incorporated file outputs versus the exact same system with the file outputs removed, this effect could be observed. No data on the effect of file input operation on these times was derived since all of the models examined included only file output operations. However, it is felt that the results would be similar.

### 3.3 Bus Resolution Functions

The next issue examined was the effect of bus resolution functions on simulation times. Bus resolution functions are commonly used within VHDL to allow for the abstraction of different bus protocols as well as different interconnection topologies. Examples of such bus resolution functions include: wired-and, wired-or, and various handshaking protocols. Currently, the ADEPT environment uses a two way, four state fully interlocked handshake on a single signal. This handshake protocol is implemented using a bus resolution function in which certain states have a higher priority over others and can overwrite these states when assigned to the same signal. This implementation allows the ADEPT system to use a signal with one resolved status field to pass tokens between modules.

A two wire handshake system, which does not use bus resolution functions, was developed for a comparison. This type of handshake still allows for a four state fully interlocked handshake protocol, but eliminates the need for a bus resolution function. Simulations of models built with the bus resolution function implementation have been compared against the same models built using the two wire handshake architecture. By replacing this bus resolution function with a two wire scheme, the effect of VHDL bus resolution functions on simulation execution times was examined.

### 3.4 Size of a Model

Another concern with the growing use of VHDL is how it handles models with a larger number of modules and how these more complex models effect simulation execution times. Numerous users have claimed that as their models become increasingly large their simulation execution times seemed to grow superlinearly. This claim of superlinear growth in execution times prompted investigation into this issue. To determine what effect the size of a model has on the simulation execution time, a special test model was created. This model was composed in such a way that the number of modules can be increased in a measurable fashion. By developing a type of modular model, the relationship between model complexity and the resulting simulation execution times was determined.

### 3.5 Testing Procedure

To accurately determine the effect of these tests on simulation times, two different simulators were used. The use of two different simulators was needed to determine if the effect of a specific model characteristic on execution time is simulator specific. These tests are for comparison purposes only, therefore, the names of these two simulators will remain anonymous. The simulators will be referred to as simulator A, and simulator B throughout this paper.

To develop these tests each simulator was run in batch mode using the Unix *time* command. This scheme allowed for each simulation run to be timed during its execution. The tests were all run on one Sun station SPARC 10 fitted with dual processors and 128 megabytes of memory. The tests were run during off peak hours to guarantee near exclusive use of the machine for these trials. Small discrepancies seen in some of the data taken can be attributed to other processes being executed during these tests. Using the Unix *time* command, three separate times are displayed: real, user, and system. The user times were collected for this experiment, which correspond to the amount of time the specific processes spent in the system. Note that this is not the wall clock time which is referred to as the real time with the "time" command. To ensure a more reliable test, it was decided that each simulation should be run at least three times. Three different lengths of simulation were also chosen resulting in a total of nine simulations for every VHDL model.

Two different ADEPT models were used to test the effect of signal size on simulation times. The first model chosen was the Algorithm to Architecture Mapping Methodology

(ATAMM) [2]. This model consists of slightly more than 200 ADEPT library modules connected by a total of 196 signals. The function of this model is to represent a seven node arbitration graph. The ATAMM model essentially generates a token every two nanoseconds at the input which in turn corresponds to a signal being generated at the model input every two nanoseconds. This model was first timed using the standard ADEPT modules which contain eighteen tag fields, and one status field. Since the ATAMM used only two of these tag fields for its execution, the tag field size was modified to contain various sizes of tag fields ranging from two to eighteen. Only the size of the tag field was reduced. Since the majority of the tag fields in this model were unused, the function of the system level model was unchanged.

The second model tested was the model of a stream memory controller (SMC) [3]. This model is composed of approximately 1,500 ADEPT modules. This model used a minimum of 10 tag fields allowing the number of tag fields to be varied from 10 to 18. By using two different models more specific conclusions about the effect of signal size on simulation execution times can be drawn.

To determine whether file input and output have any effect on the VHDL simulator execution times, a second test was developed. This test compared the simulation execution times of the ATAMM, both with and without file I/O. To eliminate file output from this model one of the ADEPT primitives was recoded. This recoding did not affect the model in any way except to eliminate write statements from the VHDL code.

A test to show the effect of the bus resolution function on simulation execution times was developed. The ADEPT modules had to be recoded to support a two wire handshaking protocol rather than using the standard ADEPT bus resolution function. Once the modules were recoded, the ATAMM model was modified to use these new modules. Once this model was created using the two wire handshaking primitives, the number of tag fields was also changed to determine if the size of a VHDL signal affected simulations using the two wire handshaking and the bus resolution function in the same manner.

A test to compare model complexity and size, versus execution time was developed. A modular type of model was developed in order to allow the size and complexity to be increased in a measurable fashion. A model of an N stage linear pipeline was used for this experiment. By using the linear pipeline, the number of stages can be altered allowing the size of a model to be changed without affecting the overall functionality of the design. The number of stages in the pipeline corresponds directly to the total number of VHDL signals within the model. Several models ranging from twenty to four hundred pipeline stages were examined for this test.

The data presented here is only a sample of the actual data that has been collected refer to [4] for the entire data set.

## 4. Results

### 4.1 Reduction of Signal Size

Three sets of results were generated for each model. For the ATAMM model, the three simulation lengths chosen were: 1,000, 10,000, and 100,000 ns. A sample of these graphs can be seen in Figure 1. The simulation execution times with respect to the number of tag fields were taken. In addition, the results of each simulator were placed on the same graph to give a comparison between the two. A best fit linear regression line is also drawn in for each data set.

Figure 1 shows the effect of tag field reduction versus simulation execution times for the ATAMM Model with a simulation length of 10,000 nanoseconds. Using this model and input configuration, simulator A exhibited a speedup factor of 1.80 when the tag fields were changed from eighteen down to two; while simulator B yielded a speedup of 2.52. Simulator A, proved to be 2.66 times faster than simulator B with only two tag fields. This ratio, however, steadily expanded to 3.72 for the original token size of eighteen tag fields. Thus simulator A, on average, demonstrated a decrease in simulation execution time of 20.13 seconds per tag field eliminated, while simulator B showed an average decrease of 101.01 seconds per tag field reduced.

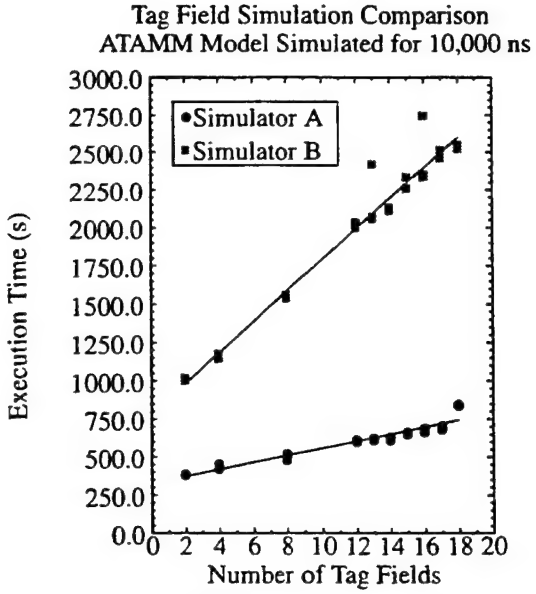


**Figure 1. Tag Field Simulation Comparison for ATAMM Model**

The effect of signal size versus simulation execution time was also examined using the SMC Model. This model was considerably larger than the ATAMM model and contained roughly eight times the amount of VHDL signals. It should be noted that since the SMC utilized a larger portion of the tag fields, the tag field size could only be reduced from eighteen down to ten. The simulation lengths chosen for this model, 1350,000, 270,000, and 540,000 ns correspond to actual simulation times needed to complete different applications on the SMC model. Results of the tag field reduction versus execution times for the SMC model with a 135,000 ns simulation time can be seen in Figure 2.

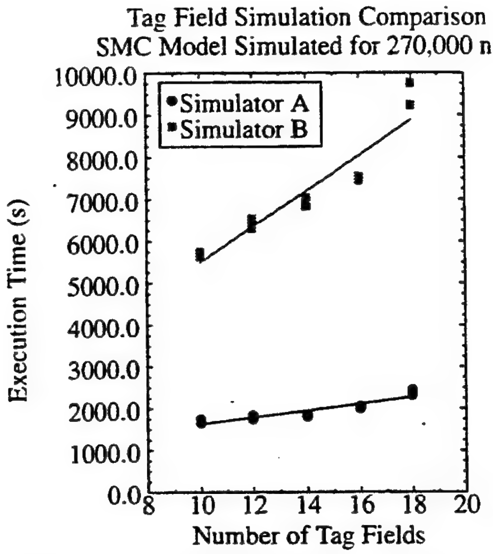


**Figure 2. Tag Field Simulation Comparison for SMC Model**

In this case simulator A demonstrated a speedup factor of 1.40, while simulator B produced a speedup of 1.65. Simulator A also finished an average 3.36 times faster than simulator B with only ten tag fields. However, this ratio grew to 3.97 for the full eighteen tag fields. These results give simulator A an average decrease in simulation execution time of 84.3 seconds per tag field eliminated, while simulator B has an average decrease of 464.9 per tag field removed. For all of this data it should be noted that the reduction in tag fields is a direct correlation to the reduction of signal size in a VHDL simulation. Each tag field that is eliminated corresponds to the removal of one integer element of an array within each signal.

Several conclusions can be made after examining the results of the signal size versus simulation lengths studies. The first point is that in all of these graphs the relationship between signal size and simulation execution times is linear. Therefore, no matter which simulator is chosen, the average simulation execution time can be significantly decreased by reducing the size of the token's tag field (the signal). For the ATAMM model, simulator A, on average, resulted in a factor of 1.83 in speedup when the standard eighteen integer tag field token was reduced to the minimal two integer tag field token. The speedup results for simulator B were even better than those of A, resulting in an average overall speedup of 2.53. However, even though simulator B resulted in higher speedup factor than that of A, its performance was still considerably slower. As shown by the graphs, simulator B was at least 2.67 times slower than that of A. Unfortunately, this ratio grew even larger as the size of the signal increased. Therefore, one must be very careful in choosing the simulator that is used.

Analogous results for the SMC were also seen. An average speedup factor for simulator A of 1.43 was obtained when reducing the full eighteen integer signal size down to ten. Simulator B produced an average speedup factor of 1.63 when comparing its reduced signal size to the standard ADEPT signal size. It must be noted that the difference between the SMC and ATAMM results are due to the fact that the SMC model used a larger number of the available tag fields thus restricting the amount by which the tag fields may be reduced. However, the overall results between the two different models are very similar. As was the case with the ATAMM model, the ratio of simulator B to simulator A grew as the size of the signal was increased for the SMC model.The overall conclusion is that not only is simulator A on average faster than B, but it is also able to handle larger signal sizes more efficiently.

### 4.2 File I/O Comparison

To perform this comparison the ATAMM model was again run for three different simulation lengths without file I/O included. The average of three execution times for simulations run at each of these lengths was taken. The number of tag fields was also varied for these tests in order to show the existence of any correlation between signal size and file input and output. Sample results are shown in Figures 3 and 4. The result of this experiment was that the addition of file output did not seem to have a significant effect on the simulation execution times. However, it is also apparent from the approximately parallel lines on these figures that there is an associated fixed overhead involved with file output.

The results show that by removing the file outputs, which correspond to 64.64 Megabytes of output data generated by invoking the VHDL write command over 400,000 individual times, a small constant amount of speedup was gained in the simulation execution times. The amount of speedup depends on the specific model and the amount of file I/O operations it performs, and the total length of the simulation being tested. Again, the amount of speedup is simulator dependent.

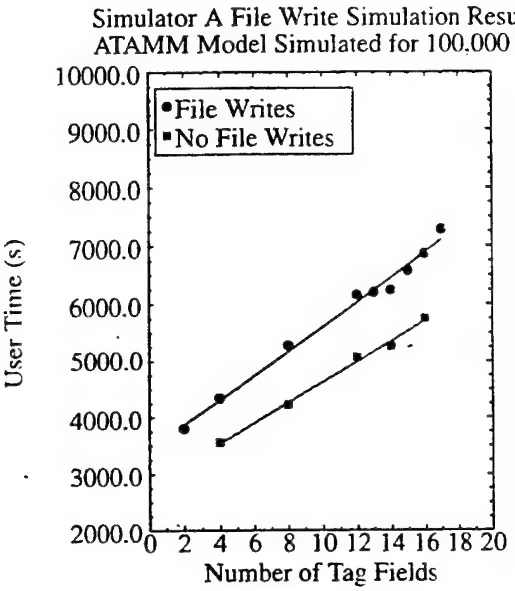


Figure 3. File Input and Output Test for Simulator A

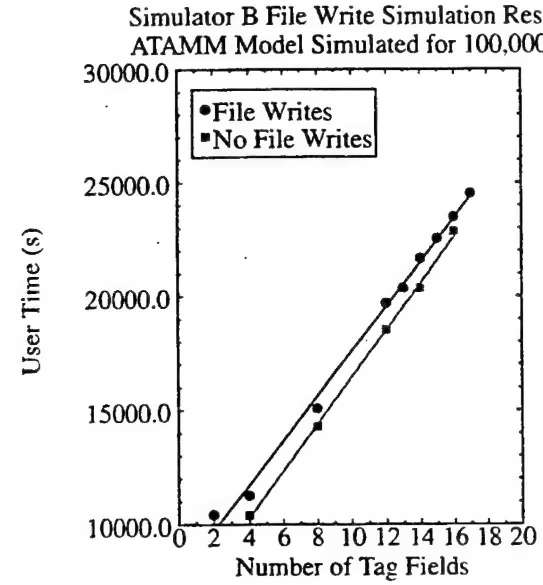


Figure 4. File Input and Output Test for Simulator B

### 4.3 Bus Resolution Versus Two Wire Handshaking

Because of the extensive modifications required to perform this test, such as constructing a new set of ADEPT primitives which utilized a two wire handshake rather than the bus resolution function, only the ATAMM model has been tested with this modification. The number of tag fields was also varied to allow the interaction of the bus resolution function along with the signal size to be monitored. These tests were run for the same simulation lengths as before for the ATAMM model: 1,000, 10,000, and 100,000 ns. Figure 5 presents the simulation execution times for both the two wire and bus resolution functions for the ATAMM model with a simulation length of 100,000 ns.

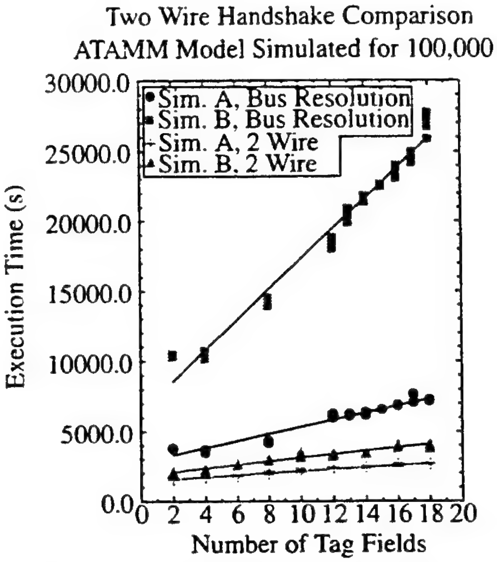


Figure 5. Bus Resolution Versus Two Wire Handshake

As shown in Figure 5, the average speedup obtained on simulator A using a two wire handshaking protocol was 2.7, while for simulator B, the average speedup was 6.5.

### 4.4 Model Complexity

By creating a model of an N stage linear pipeline within the ADEPT environment the effect of simulator execution time versus model complexity could be observed. Using the ADEPT environment allows for the creation of this linear pipeline which is a series of buffered delay elements. The modular framework of the ADEPT environment also allows for the number of stages in the pipeline to be altered rather easily. Each stage in the pipeline example corresponds to a 1 ns fixed delay module followed by a buffer module. Therefore, to construct a twenty stage linear pipeline twenty of these fixed delay-buffer pairs were strung together.

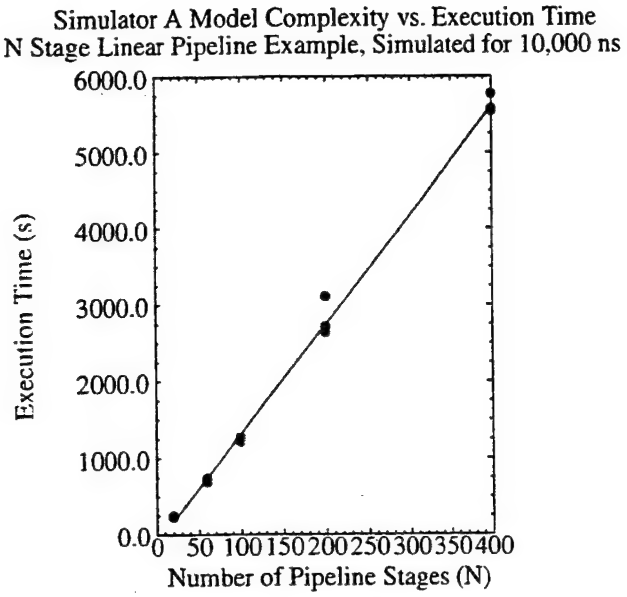


**Figure 6. Model Complexity Test for Simulator A**

Figures 6 and 7 present the results of this experiment. These figures show that as the size of the model is increased the simulator execution time also increases in a very linear fashion. The actual slope in Figure 6 is 14.31 with a standard error of 0.18. These results do not show that as the size of a model increases, the simulator execution time increases superlinearly.

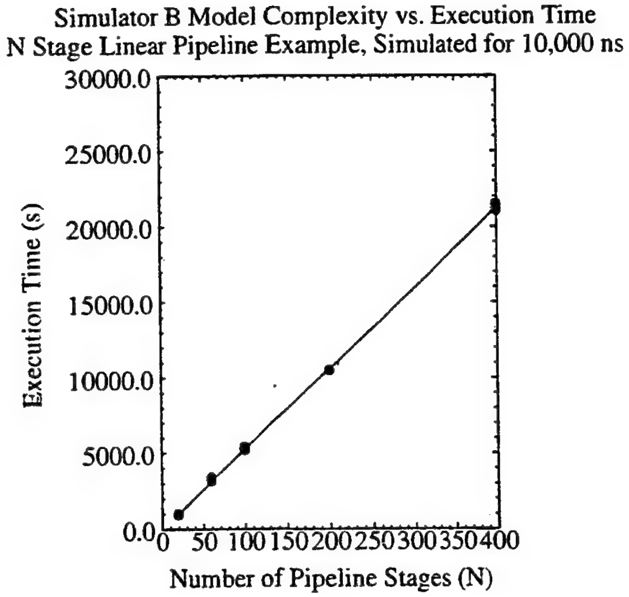


**Figure 7. Model Complexity Test for Simulator B**

A slope of 53.26 with a standard error of 0.21 resulted for simulator B. When compared to the slope for Figure 6 (the corresponding result for simulator A), the differences between the two simulators can again be noted.

## 5. Conclusions

This paper presented the results of tests that were conducted to determine the effect of VHDL system level model characteristics on simulation execution times. Significant insight into what can be done within the model to decrease the total time for simulation of complex VHDL models has been gained. For the models tested the size of the signal has a very linear effect on execution times. Even though most of this signal was not being used within the models, it still presented significant overhead to the simulator. This overhead results from the simulator passing around the full signal of an array of integers. By reducing the size of the signal by a factor of 1.8-6.33, the average speedup factor of 1.43-1.83 or 1.63-2.53 was obtained depending on which simulator was used.

The file I/O experiment showed that by removing file writes within a design, the user can reduce, for a given model and simulation length, the execution time of a simulation by a constant factor. Although this constant might not be significant for small designs or short simulation times, effects may be significant for larger and more complex models.

The use of bus resolution functions was shown to have a significant effect on simulation time. This single factor resulted in decreased simulation execution times by a factor of 2.77 or 6.43 depending upon which simulator was used.The results also showed that by replacing the bus resolution function, the execution times of the two simulators were comparable. This one experiment illustrated the significant differences in various simulators.

It has been conjectured that as the size and complexity of models increase, the simulator execution times grow superlinearly. It was shown through these experiments that the number of VHDL signals has a very linear effect on the simulation execution time. However, the slopes of these lines between the two simulators were significantly different. For example, these results show that simulator A can handle VHDL models with an increased number of signals more efficiently than simulator B.

The data taken to date has given considerable insight into what can be done to speed up these simulation execution times. Several different aspects of VHDL code were examined and the results have shown that careful planning of VHDL code can greatly reduce simulation execution times. Not only do these results show different methods by which to reduce execution times but they also show that the choice of simulators plays an important part in the simulation execution time and the effect of these methods on the execution time. This more detailed understanding of the simulation environment allows future VHDL users to build and simulate models in a more efficient manner.

## Acknowledgments

A number of CSIS students and staff have contributed to the development of the ADEPT environment and to the material presented in this paper. These students include Bob McGraw, Ramesh Rao, Gnanasekaran Swaminathan, Charles Choi, Maximo Salinas, and Moshe Meyassed.

## References

[1] Eric D. Cutright, Ramesh Rao, Sanjay Srinivasan, Barry W. Johnson, James H. Aylor, Ronald D. Williams. *The Advanced Design Environment Prototype Tool (ADEPT) System: A User's Manual With Tutorial.* Technical Report #910916.0, Department of Electrical Engineering, University of Virginia, September 1991

[2] Eric D. Cutright, Ramesh Rao, Barry W. Johnson, James H Aylor. *Modeling an ATAMM-Based Multiprocessor System Using VHDL.* Technical Report #910111.0, Department of Electrical Engineering, University of Virginia, January 1991.

[3] Sally A Mckee. *Maximizing Memory Bandwidth for Streamed Computations.* Ph. D. Dissertation, Department of Computer Science, University of Virginia, 1995.

[4] .Andrew P. Voss, Bob H. Klenke, James H.Aylor. The *Analysis of Modeling Styles for System Level VHDL Simulations,* Technical Report #950719.0, Dept. of Electrical Engineering, University of Virginia, July 1995.

**Appeared in the Proceedings of the Fall VHDL International Users Forum, 1995, pp. 1.7-1.13**

# Performance Modeling of Multicomputer Systems in VHDL using ADEPT

Robert H. Klenke, Andy Voss, James H. Aylor

Center for Semicustom Integrated Systems
Department of Electrical Engineering
University of Virginia
Charlottesville, VA 22903

## Abstract

*This paper describes the implementation of a performance modeling environment for multicomputer systems. This environment is based on the ADEPT performance modeling environment developed in the Center for Semicustom Integrated Systems at the University of Virginia. ADEPT is an integrated performance and dependability modeling environment that uses the IEEE Std. 1076 VHDL language for simulation.*

*A library of generic communication modules that can be parameterized to model different communications protocols and topologies was developed for ADEPT. Five different existing communications networks were modeled as examples of the library: Asynchronous Transfer Mode (ATM), Ethernet, Myrinet, Raceway interconnect, and the Scalable Coherent Interface (SCI). The new ADEPT communications modules allow users to create performance models of multicomputer systems in a more timely and efficient manner.*

## 1. Introduction

As multicomputers become less and less costly, they are being applied in the area of embedded processing and control. In these applications, it is very important to ensure that the architecture selected, in terms of the number of processors, the type and topology of the interconnect, etc., meets the performance requirements. Furthermore, in order to avoid costly redesigns, it is important that performance evaluation be done as early in the design cycle as possible, even before all of the system functionality is defined.

As an example, the RASSP (Rapid Prototyping of Application Specific Signal Processors) program is focusing on obtaining a 4X improvement in the design cycle time, cost and quality of embedded digital signal multiprocessors for DoD applications [1]. There is a heavy emphasis on performance modeling of these DSP systems very early in the design cycle in the RASSP program in order to meet these goals.

This paper presents a performance modeling environment for multicomputer networks that was developed under the RASSP program. This environment was added to an existing performance modeling tool developed at the University of Virginia called ADEPT.

The remainder of the paper is organized as follows: section 2 presents a brief overview of the ADEPT system. Section 3 presents the communications library added to ADEPT to facilitate the modeling of multicomputer systems. Section 4 presents some results of verifying the performance models constructed with this library. Finally, Section 5 presents some conclusions.

## 2. The ADEPT System-level Modeling Environment

ADEPT (Advanced Design Environment Prototype Tool) is a unified end-to-end design environment developed in the Center for Semicustom Integrated Systems at the University of Virginia [2]. ADEPT supports both system level performance and dependability analysis in a common design environment using a collection of predefined library elements. ADEPT also includes the capability to simulate both system level and implementation level (behavioral) models in a common simulation environment. This capability allows the stepwise refinement of system level models into implementation level models.

Two approaches to creating a unified design environment are possible. An evolutionary solution is to provide an environment that "translates" data from different models at various points in the design process and creates interfaces for the non-communicating software tools used to develop these models. With this approach, users must be familiar with several modeling languages and tools. Also, analysis of design alternatives is difficult and is likely to be limited by design time constraints.

The approach being developed in ADEPT is to use a single modeling language and mathematical foundation which decreases the need for translators and multiple models, thus reducing inconsistencies and the probability of errors in translation. Additionally, the existence of a mathematical foundation provides an environment for

---

**The authors would like to acknowledge the support provided by the Advanced Research Projects Agency under contract number F33615-93-C-1313.**

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

complex system analysis using analytical approaches.

Simulators for hardware description languages accurately and conveniently represent the physical implementation of digital systems at the circuit, logic, register-transfer, and algorithmic levels. By adding a system level modeling capability based on extended Petri Nets and queuing models to the hardware description language, a single design environment can be used from concept to implementation. The environment allows for the mixed simulation of both *uninterpreted* (performance) models and *interpreted* (behavioral) models due to the use of a common modeling language.

ADEPT implements an end-to-end unified design environment based upon the use of the VHSIC Hardware Description Language (VHDL), IEEE Std. 1076 [3]. ADEPT supports the integrated performance and dependability analysis of system level models and includes the capability to simulate both uninterpreted and interpreted models in a common simulation environment using a technique called *hybrid modeling*. Hybrid modeling allows the stepwise refinement of system level models into implementation level models. ADEPT also has a mathematical basis in Petri Nets thus providing the capability for analysis through simulation or analytical approaches [4].

## 2.1 The ADEPT Modules

In the ADEPT environment, a system model is constructed by interconnecting a collection of predefined elements called ADEPT modules. The modules model the information flow, both data and control, through a system. Each ADEPT module has a VHDL behavioral description and a corresponding mathematical description in the form of a colored Petri Net (CPN) based on Jensen's CPN model [5]. The modules communicate by exchanging *tokens*, which represent the presence of information, using a fully interlocked, four-state handshaking protocol [6]. The basic ADEPT modules are intended to be building blocks from which useful modeling functionality can be constructed. In addition, custom modules can be developed by the user if required and incorporated into a system model as long as the handshaking protocol is adhered to. Finally, some libraries of application-specific, high-level modeling modules such the communications network modeling library described in this paper have been developed and included in ADEPT.

ADEPT tokens are implemented as a VHDL record structure. In the token, the two most important fields are the *STATUS* field and the *COLOR* field. The STATUS field is used to implement the token passing mechanism; that is, the "handshaking" between the ADEPT modules. The COLOR field is an array of integers that hold user-specified information. Modules are provided which can manipulate the information in the COLOR field.

The set of basic ADEPT modules is divided into six categories: *control* modules, *color* modules, *delay* modules, *fault* modules, *miscellaneous* parts modules, and *hybrid* modules. The control modules are used to manipulate the flow of tokens in a model. A majority of the control modules have been adapted from Dennis [7]. ADEPT modules in the color and delay categories enable the manipulation of the token color and model temporal aspects of a system, respectively. The fault modules are used to model the presence of faults and errors in a system model for dependability analysis. The miscellaneous modules are modules that perform data collection with the ADEPT system. Hybrid modules aid in the construction of hybrid models. A more detailed description of the entire ADEPT module set can be found in [8].

## 2.2 The ADEPT Tools

The ADEPT system is available on Sun platforms using Mentor Graphics' *Design Architect* as the front end schematic capture system, or on Windows PCs using OrCAD's *Capture* as the front end schematic capture system. The overall architecture of the ADEPT system is shown in Figure 1.

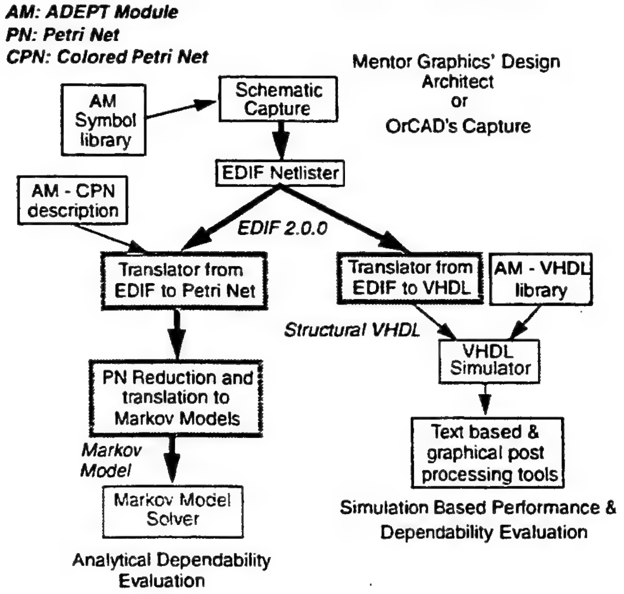

AM: ADEPT Module
PN: Petri Net
CPN: Colored Petri Net
Mentor Graphics' Design Architect or OrCAD's Capture
AM Symbol library
Schematic Capture
EDIF Netlister
AM - CPN description
EDIF 2.0.0
Translator from EDIF to Petri Net
Translator from EDIF to VHDL
AM - VHDL library
Structural VHDL
VHDL Simulator
PN Reduction and translation to Markov Models
Markov Model
Markov Model Solver
Text based & graphical post processing tools
Analytical Dependability Evaluation
Simulation Based Performance & Dependability Evaluation

**Figure 1. ADEPT design flow**

The schematic front end is used to graphically construct the system model from a library of ADEPT module symbols. Once the schematic of the model has been constructed, the schematic capture system's netlist generation capability is used to generate an EDIF (Electronic Design Interchange Format) 2.0.0 netlist of the

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

model. Once the EDIF netlist of the model is generated, the ADEPT software is used to translate the model into a structural VHDL description consisting of interconnections of ADEPT modules. The user can then simulate the structural VHDL that is generated using the compiled VHDL behavioral descriptions of the ADEPT modules to obtain performance and dependability measures.

In addition to VHDL simulation, a path exists that allows the CPN description of the system model to be constructed from the CPN descriptions of the ADEPT modules. This CPN description can then be translated into a Markov model using well known techniques and then solved using commercial tools to obtain reliability, availability, and safety information.

Figure 2 is an illustration of the construction of a schematic of an ADEPT model using Design Architect. The schematic shown is that of a Mercury Raceway Multicomputer system described in Section 4. Most of the elements in this top-level schematic are hierarchical, with separate schematics describing each component. The most primitive elements of the hierarchy are the ADEPT modules.

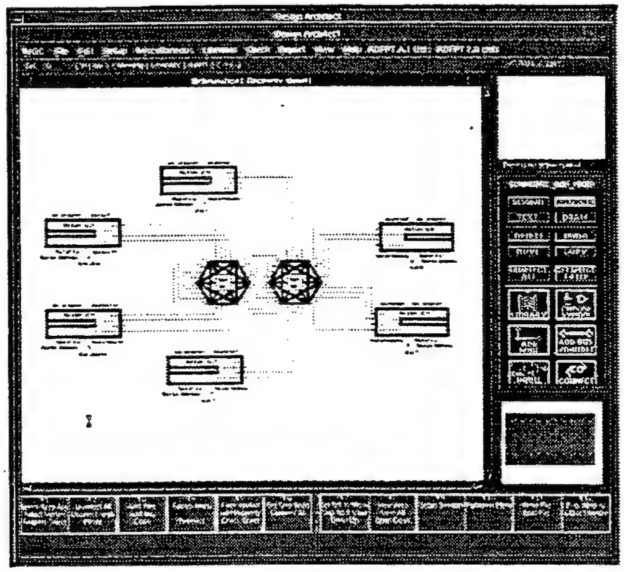


**Figure 2. Sample ADEPT Design Architect schematic**

## 3. ADEPT Multicomputer System Modeling Modules

It is possible to model a multicomputer system in ADEPT using the predefined ADEPT modules. However, as these modules are at a very low level of functionality in terms of token routing and tag field manipulation, the multicomputer system models constructed with them tend to require a large number of modules. An ADEPT model with this many modules usually becomes hard to construct, analyze, and debug and also requires a significant amount of time to simulate because of the large number of signals between the individual modules [9].

To make the construction and simulation of multicomputer system models more efficient, a library of modules that were targeted to this specific application was added to the ADEPT system. In order to design this "communications library," example communications networks were selected and analyzed. This analysis was used to design modeling modules and a communications token structure that was as generic as possible to allow reuse of the modules and facilitate future expansion of the library. Five state-of-the-art communications networks were analyzed; Asynchronous Transfer Mode (ATM), Scalable Coherent Interface (SCI), Ethernet, Myrinet, and the Mercury Computer Raceway Crossbar Switch

After examination of the five different communications networks it was then necessary to decompose their protocols into a set of generic functional blocks that could be added to the ADEPT environment. Before this decomposition colud be performed, a common token structure had to be defined so that these library elements could communicate with each other and share the same token format. This section first describes the token structure designed for the communications library elements. After the token structure is defined, the new library elements are presented.

### 3.1 The Communications Library Token Structure

The communications library uses the normal ADEPT token structure. However, there is some basic information that must be passed in a token that models a packet of data in a communications network such as the destination address and the size the a message. Therefore, it was decided that within the communications library some of the tag fields would be reserved for use by these modules only.

All five of the protocols studied support different message types. Therefore, the type of message must be specified in the token. Between the five protocols, there are numerous message *types*. However, the most basic types are a set-up, a general data, and a tear-down type. By providing this information within the token, the communications library elements can determine what kind of response is necessary according to the type held within the token.

The five protocols also support many various transactions across their networks. Examples of these transactions are: send, request, voice, and write. These transactions determine what the receiving node is supposed to do with the data that it is currently receiving. Therefore,

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

the second piece of necessary information is the *transaction* type.

In order for the communications library to properly route messages between different nodes of a network, the token must convey routing information. In the various protocols, this routing information is usually conveyed in the packet header of the message. The *route* field within the token will take care of routing each individual token from its source to the correct destination address.

To properly model the delay of the various networks topologies, these modules must know the size of the message being transmitted. Without providing the size of the individual messages being transmitted across a network, only a lump message delay could be introduced in the model. This would mean that every message being passed through the network would incur the same delay no matter what its size. This type of delay would be unacceptable in these models. Consequently, message *size* is important information that must be included into the reserved tag fields.

Three of the communications protocols, ATM, Raceway, and SCI all support a message priority scheme. This information allows messages marked with a higher priority to have precedence over lower priority messages. Although priority is handled differently within each of these protocols, there still exists a need to mark different priority levels on messages being passed through a network. Hence, another parameter to include into the token is the message *priority*.

The Ethernet and the SCI protocol include the source address of the message in their message headers. This source address notifies receiving nodes of their messages origin. Thus a *source* field was also needed within the communications library token.

Two of the communications protocols, Raceway and the SCI, allow remote memory operations to be performed across the network. They accomplish this operation by including an optional address field within their message structure. This optional field corresponds to the memory address at which these operations are to be performed. Although it is an optional field within the message protocols, the *address* is used for certain transactions and was therefore included.

The last piece of information needed within the communications library token structure is a *data* field. This field represents the actual data being passed. It is important to note that in performance modeling, the actual data passed through the network is usually not important in determining the network performance metrics. However, reserving a tag field in the token for user defined data increases the flexibility of the modeling environment.

### 3.1.1 Tag Field Breakdown

After determining the number of tag fields that would be used in the communications library the information needed to be placed in specific tag fields. Since there were seven different items that needed to be represented in the tag fields, this would normally require the use of seven separate tag fields. However, in [9], it was shown that the number of tag fields used in a model has a large effect on the simulation time of an ADEPT model. Therefore, some of these items were grouped together or compressed into a single tag field. This grouping allowed the communications library modules to function with a minimum of five tag fields. In order to determine which items would be packed together each information category was enumerated to determine how many values it could take on.

The *priority* information for any given protocol only has four different values it can take on. The message *type* has only eighteen possible values. There are twenty four different *transactions* that can be assigned to a given message. Therefore, these three information categories were compressed into a single tag field using one or two digits in the integer tag per item. The collection of these three fields was placed into tag field one of the standard ADEPT token and is referred to as the **ID** tag field.

The *size* of a message for any of the protocols has no limit other than the maximum packet size that will be transmitted at any given time across the network. However, there is no restriction on the maximum message *size* a CPU can send. If a message larger than the maximum packet size is given to any of these networks, they simply break this message up into several packets that correspond to the maximum packet size of the given network. Since the size of a message is essentially unbounded, an entire tag field was reserved to hold the message size information. Thus, tag field two has been reserved to hold the *size* of the message in bytes. Tag field two is referred to as the **size** tag field.

The third tag field was chosen to represent the *route* information for the messages passing through the network. For the case of the Ethernet and the SCI where the *source* information is also incorporated into the message, this tag field will also incorporate this source category for these two protocols. Digit compaction was also used in this case when the *source* and *route* information are both incorporated into one tag field. For the SCI and Ethernet examples the *route* information is represent in digits one through five of tag field three and the *source* information is represent in digits six through ten of this tag field. For the other communications protocols where the source information is not used, the route information uses all ten digits of the tag field. Tag field three is referred to as the **path** tag field. Depending on the communications protocol being modeled,

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

this field can contain route information or source and route information. To help the network determine what information this field is representing, the tag field will be set to either negative or positive depending on whether it incorporates the *source* and *route* information or solely the *route* information respectively.

The *address* information contained in the SCI and Raceway protocols was implemented in tag field four of the communications library token structure. This tag field is referred to as the **address** tag field.

The last tag field, tag five, represents the user specified *data*. This tag field is unused by any of the communications modules and is reserved solely for the purpose of allowing the user one reserved tag field to hold user-defined data. Tag five is referred to as the **data** tag field.

Each of the five tag fields have an alias associated with them to make referring to them within the library modules easy on the user. These names are coded in as constants within the communications library package so that the user never has to worry about referencing a specific tag field number. Table 1 provides a summary of the names for each tag field and the information that they represent.

**Table 1. Communications Library Tag Field Implementation**

| Tag Field | Implementation Name | Information Represented |
|---|---|---|
| tag1 | ID | *priority, type, transaction* |
| tag2 | size | size |
| tag3 | path | *route* and in some cases *source* |
| tag4 | address | *address* (not used in some models) |
| tag5 | data | user defined *data* |

### 3.2 The Communication's Library Modules

The different communications protocols had to be broken down into generic modules that could be incorporated into the ADEPT framework. The modeling of the communications protocols was broken down into three basic elements; a transmitter, receiver, and router. Each of these elements may be different for the varying protocols (such as the specific bus router for the Ethernet model) but their overall functionality remains the same. In addition, a simple CPU has also been included to allow complete multicomputer models to be constructed. A more complex CPU model that incorporates the appropriate interface can be used.

#### 3.2.1 Transmitter Library Elements

The transmitter library element is responsible for accepting an input message from the CPU model, breaking the message into its appropriate parts according to the particular protocol being modeled, and sending it across the network via the router modules. Since the execution of each protocol is different, five separate transmitter elements were constructed for the communications library. All of the transmitter elements have three different generic properties associated with them. The first generic property is the *routefile* which specifies the file from which the transmitter will get its routing information. The second generic property is the *source_ address* property. This is used by the transmitter so that it can place its source information within the path tag fields for the case of the Ethernet and the SCI communications protocols. All the transmitters have this property to assist the user in determining the proper routing of tokens to specific destinations. The final generic property common for all five of the transmitters, is the *max_size* property. This property specifies the maximum individual packet size in bytes that this particular protocol can transfer across the network. The *max_size* parameter is set to the correct value for each different communication protocol so that all the user really needs to specify is the source address and path to the route file.

Each transmitter behaves slightly differently according to the communications protocol it is modeling. However, the basic function of the transmitter modules starts with it receiving a token on its input from the CPU. Upon receiving the token, the transmitter then looks up the route file from the generic route path property. This route file is basically a translation table where the user specifies the destination address of all the given nodes in the network and the route path that is needed to send a message from this particular node to the nodes listed in the table.

After getting the route information from the route file and placing this information into the path tag field, the transmitter sets up the communication path. Once a communication path has been established, the transmitter breaks up its input message according to its size and the maximum packet size generic for each model. The transmitter then sends individual tokens across the network to the destination address. Each token that is transmitted is considered a data packet and the different transmitter elements set their corresponding type of packet data. After all the packets have been sent the last packet transmitted across the network is a clear or tear-down path message. This packet contains the total message sizes and signals the corresponding receiver that an entire message has just been sent. Once this final tear-down token has been acknowledged the transmitter then acknowledges the token

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

at its input to the CPU to signify that this message has been transmitted across the network.

The ADEPT symbol for the Raceway transmitter module is shown in Figure 3.

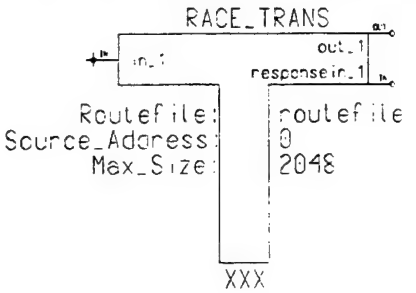


**Figure 3. Raceway transmitter module**

### 3.2.2 Receiver Library Elements

The receiver elements function as the converse of the transmitter elements. The receiver takes all of the individual packet tokens and reconstructs the original message to be sent on its output to the CPU. Each communications protocol also has its own individual receiver but the basic functionality of these various receivers is the same. Upon receiving the set up token, the receiver acknowledges this token to signify that the path has been set up. As the packet tokens are sent, the receiver acknowledges each packet until the tear-down token is reached. When the tear-down packet is received, the receiver sets all of the appropriate tag fields according to the total message that has just been received and sends this token to its output. This output token now contains all of the original token information that was present at the input of the transmitter module before the network communication took place.

Some of the receiver modules have a generic *length* property listed on the symbol. This specifies the size of the queue at the input to the receiver module. A queue is used in the receiver to model the fact that in the actual network, the receiving of the message is decoupled from the CPU.

The ADEPT symbol for the Raceway receiver module is shown in Figure 4.

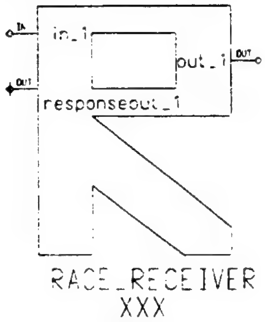


**Figure 4. Raceway receiver module**

### 3.2.3 Router Library Elements

The router modules are responsible for taking a token on any of their inputs, and according to the path tag field, routing this token to the specified output port. There are four different size routers in the library: a two, four, six and an eight port router. Each of these routers is a fully connected router supporting as many simultaneous transactions as it has ports.

Three generic properties are provided on the router modules. The first of these is the hardware *delay* parameter. Each token being routed from one input to another output will be delayed by this fixed delay. The second generic property is a Boolean variable called *oneway.* This generic indicates whether or not the router allows simultaneous transactions on a given port. With this generic set to "true," the router, once a communication path has been set up, will only allow these ports being used to support a data transfer in one direction as apposed to bidirectional. The last generic property is another Boolean variable called *pre_empt*. This variable when set to true allows any token with a higher message priority to preempt a lower priority message that it is in contention for resources with.

When a token arrives at any of the router inputs, the router looks at the path field. The least significant digit in the path field tells the router which output port this token should be routed to. After the token has been routed to its appropriate output port, the router performs a right digit shift of the path tag field so that the next router along the network path gets its correct route path. This implementation scheme allows several routers to be strung together to form a large communications network. Unlike the transmitter and receiver, the function of the router was constructed in such a way that it can be used with four of the five different communications protocols. The only protocol the router does not support is the Ethernet. Since the Ethernet is based upon a bus type of architecture where there is only one physical path and consequently only one message may reside on this path at any given time, a different implementation was needed to model this functionality.

Each one of the four communications protocols has a router module that has been incorporated into a schematic with a custom symbol for it. The generics on the router have been set to model the specific protocol. As an example, the ADEPT symbol and schematic containing the router module for the Raceway crossbar switch is shown in Figure 5.

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

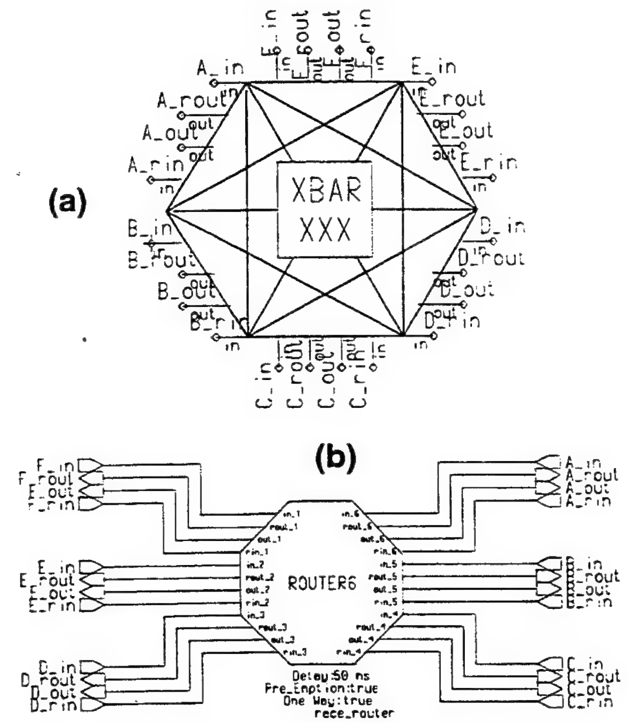


**Figure 5. Raceway crossbar switch symbol (a) and schematic (b)**

The bus router is the Ethernet equivalent of the router module. This module behaves the same as the router as far as its routing scheme and basic operation. The only difference between the two is that the bus router only allows one token to reside on the network at any given time. Hence, its operation is equivalent to that of a bus. Since the bus router was developed solely for the Ethernet, the generic *oneway* and *pre_empt* Boolean tag fields were eliminated. The generic *delay* property performs exactly as it did in the router simply delaying the placement of input tokens to their output by this specified delay interval. The ADEPT symbol for the six port bus router is shown in Figure 6.

### 3.2.4 The CPU Library Element

The CPU library element reads a program from a file in a specified format and then interacts with the network model according to this program. The CPU is intended to be a very basic element and provides only a minimal "Compute, Send and Receive" type of functionality. However, its instruction set can easily be expanded to include modeling of more complex functions. Additionally, ADEPT hybrid modeling library elements and techniques have been developed that allow fully functional ISA level CPU models to be connected to the communications library network models [10].

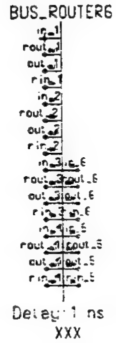


**Figure 6. Bus router module**

The CPU module has two generic parameters. The first is an integer *buff_size* which determines how many tokens the CPU can buffer on its input. The second parameter is the *filename* generic which specifies the file from which the CPU will read its program.

Presently the CPU instruction set contains seven instructions. With these seven instructions the user can send messages to other CPU's or nodes in the network, wait until a message is received from another node in the network, compute for some specified time, perform no ops for some specified time, or receive a message from its input buffer and wait a specified time from the arrival of this message, restart at the beginning of the program or stop execution of the program. The function of these instructions are outlined below:

**Sendmessg Instruction** - this instruction is responsible for sending out a token that instructs the network model to construct and send a message from this CPU to another CPU. The input parameters to the sendmessg instruction include the priority of the message, the transaction type of the message, the size of the message, the destination address of where this message is to be routed, the address tag information, and the data tag information.

**Recvmessg Instruction** - this instruction takes two parameters as its input: transaction and source address. When the CPU executes a recvmessg command, it waits until it receives a token on its input from the network model whose tag field information matches those specified as parameters in the instruction.

**Recandcmp Instruction** - this instruction is a modified version of the recvmessg command which allows the incorporation of a queued input to the CPU as well as an extra delay parameter.

**Compute Instruction** - this instruction models the execution of actual program code by the CPU. As

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

this is a simple high-level model of a CPU, the compute instruction has a delay parameter as its input. When the CPU starts a compute instruction, it simply waits for this specified delay time and continues its operation.

**No_op Instruction** - this instruction performs exactly the same as a compute instruction, but in this case, it simulates the CPU being idle for a specified period of time.

**Restart Instruction** - this is a loop instruction which starts the CPU program over at the beginning.

**Stop Instruction** - this instruction when executed by the CPU ends the CPU program.

The ADEPT symbol of the CPU module is shown in Figure 7.

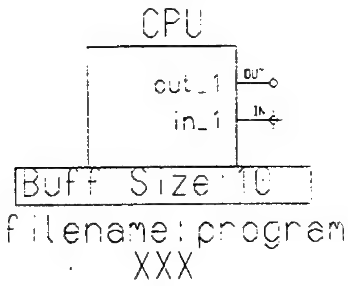


Figure 7. CPU module

## 4. Model Validation Results

This section presents the results of validating two of the communicatins protocols in the communications library, the Ethernet and the Raceway. Both of these models were validated more thoroughly than the others because additional capabilities for doing so were available. For the Myrinet, SCI, and ATM models, only the original literature that described their operation was available, so tests were run to determine that the models performed as described in the literature.

### 4.1 Results of the ADEPT Ethernet Model

The Ethernet communications library elements were validated by comparing their results against the actual hardware. By writing a program in C using Parallel Virtual Machine (PVM) [11], a real world application to validate the Ethernet communication's library elements was created. This program implemented a six node network connected via an Ethernet cable. Each node in the network was a Sun SPARC Station 10. This parallel program performed the following functions:

1.) One of the six workstations was marked as the master. This master sent a one megabyte message to each of the other five slave nodes, in a round robin fashion.

2.) After receiving their message, each slave node went to sleep for one second before sending a one megabyte message back to the master node.

3.) After receiving messages from each of the slave nodes, this sequence was then repeated.

The size of these messages was one megabyte to ensure that the communication time for one message was longer than one second. This organization guaranteed that several nodes would be trying to use the Ethernet cable at a given time, thus ensuring that many collisions and blocking conditions would occur on the Ethernet during the execution of this program. This PVM program was run multiple times yielding the data shown in Table 2. The average execution time of this C program using PVM was found to be 24.35 seconds. This produces a measured bandwidth of this Ethernet cable to be 0.82 megabytes per second. As stated in [12], the maximum achievable Ethernet bandwidth is 10 megabits per second, which corresponds to 1.25 megabytes per second. Therefore these measurements from this program are within an acceptable range.

Table 2. PVM Program Execution Times

| Trial | Time (s) |
|---|---|
| 1 | 24.10 |
| 2 | 24.01 |
| 3 | 25.26 |
| 4 | 24.85 |
| 5 | 23.93 |
| 6 | 24.77 |
| 7 | 23.51 |
| Average | 24.35 |
| Standard Deviation | 0.62 |

Using the Ethernet library elements, the same application was modeled in ADEPT. Figure 8 shows the model of this six node network as implemented in ADEPT. Each node is a hierarchical component which is composed of a CPU, an Ethernet transmitter, and an Ethernet receiver.

After generating the VHDL code using the ADEPT tools, this six node Ethernet model was simulated. The simulations run for this model showed that the last message

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

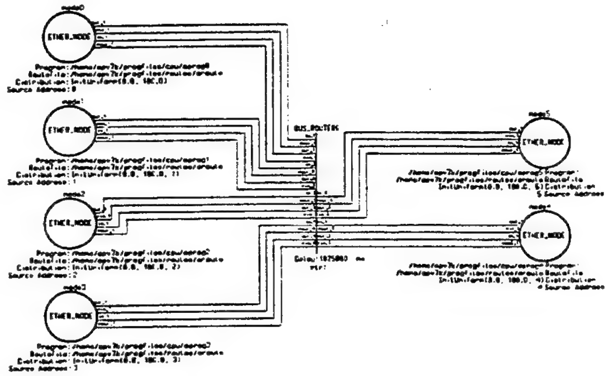


Figure 8. Ethernet model of a six node network

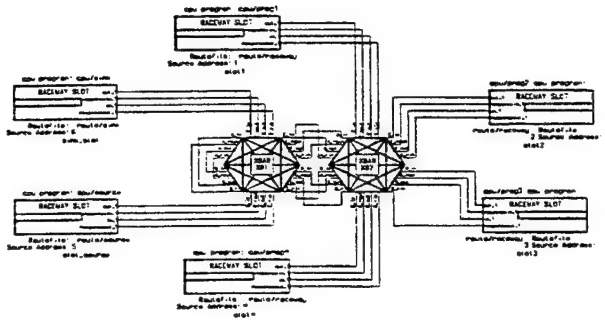
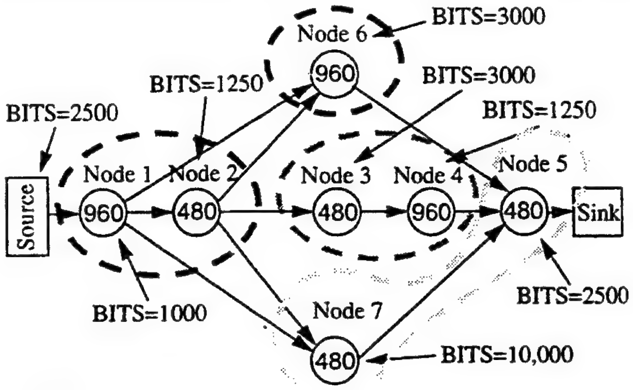


Figure 9. ADEPT model of a Raceway crossbar network



NOTES:
Node execution delays are in micro seconds
BITS=Size of messages produced by node in bytes

NODE MAPPINGS:
Node 1, Node 2: CPU1
Node 3, Node 4: CPU2
Node 6: CPU 3
Node 5, Node 7: CPU4

Figure 10. Seven node data flow graph implemented in Raceway model

was received from the network by a node after 24.39 seconds. After examination of the simulation traces it was verified that all nodes received their correct data and responded accordingly by placing their messages onto the network. Thus, the execution time of this program on the ADEPT model was found to be 24.39 seconds. When comparing this result to the average executions of the PVM program using the actual hardware, the error was found to be 0.19%. This error is small and well within an acceptable range to validate the ADEPT Ethernet communications library elements.

### 4.2 Raceway Crossbar Model Validation

Unfortunately, hardware for a Mercury Computer Raceway system was not available to the developers of the communications library. Therefore, it was not possible to validate the model of the Raceway system by comparing it to runtimes on actual hardware as was possible with the Ethernet model. However, performance models of Raceway systems have been constructed by commercial organizations and validated against actual hardware by them. These models were used for validation - a kind of *n* version programming solution.

The Raceway network used in the validation effort is shown in Figure 9. Each Raceway slot as shown within the figure corresponds to a CPU, a Raceway transmitter and a Raceway receiver similar to the Ethernet node.

The program chosen to be used in the validation effort was an implementation of an arbitrary seven node data flow graph. Four of the six Raceway slots were used the execute the arbitration graph while the fifth and sixth slot were used to source and sink the data through the arbitration graph. Figure 10 shows this arbitration graph and the allocation of the nodes to the four Raceway slots used for graph execution (the other two Raceway slots performed the source and sink functions).

After executing this program on both ADEPT Raceway model and the commercial Raceway model, two separate parameters were measured. The first parameter obtained was the time at which the first result was available at the input of the sink. From the commercial model, this first arrival time was found to be 3,407,421 nanoseconds. The second parameter obtained from this model was the interval arrival time between results once the model had reached its steady state operating phase. This interval arrival time was found to be 1,440,600 nanoseconds.

Simulating the model using the ADEPT library elements produced similar results. A summary of these results can be

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

seen in Table 3.

**Table 3. Raceway Model Result Times**

| Model Used | First Result (ns) | SS Interval Results (ns) |
|---|---|---|
| Commercial Model | 3,407,421 | 1,440.600 |
| ADEPT Model | 3.434,500 | 1,440.000 |

## 5. Conclusions

This paper has described implementation and results of a performance modeling environment for multicomputer systems based upon ADEPT. This environment models the multicomputer system at a high level of abstraction where only data routing a computation are considered. This environment allows a multicomputer system designer to quickly validate that the chosen system architecture can meet the required performance goals.

Although the library only contains elements for modeling five different communications network types; ATM, Myrinet, Ethernet, SCI, and Raceway, the methodology can easily be extended to model other types of networks as they are developed.

## 6. References

[1] Richards, M. A., A. J. Gadient, G. Frank, R. E. Harr, "The RASSP Program: Origin, Concepts, and Status," *Journal of VLSI Signal Processing*, Winter 1997, (to appear).

[2] Kumar, S. R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, R. Waxman, "ADEPT: A Unified Environment for End-to-End System Design," in *High-Level System Modeling: Specification and Design Methodologies*, Kluwer Academic Publishers, 1996, pp. 55-82.

[3] IEEE, *"IEEE Standard VHDL Language Reference Manual,"* New York, NY, IEEE Std. 1076-1993, June 6, 1994.

[4] J. H. Aylor, R. Waxman, B. W. Johnson, R. D. Williams, "The Integration of Performance and Functional Modeling in VHDL," in *Performance and Fault Modeling with VHDL*, J. M. Schoen (Ed.), Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 22-145.

[5] K. Jensen, "Colored Petri Nets: A high level language for system design and analysis," in *High-level Petri Nets: Theory and Application*, K. Jensen and G. Rozenberg (Eds.), Berlin: Springer-Verlag, 1991, pp. 44-119.

[6] F. T. Hady, *A Methodology for the Uninterpreted Modeling of Digital Systems in VHDL*, Master's Thesis, Dept. of Electrical Engineering, University of Virginia, January 1989.

[7] J. B. Dennis, "Modular, Asynchronous Control Structure for a High Performance Processor," *ACM Conference Record, Project MAC*, Massachusetts, 1970, pp. 55-80.

[8] *ADEPT A.1 Library Reference Manual*, CSIS Technical report 960625.0, University of Virginia, June 6, 1996.

[9] Voss, A. P., R. H. Klenke, J. H. Aylor, "The Analysis of Modeling Styles for System Level VHDL Simulations," *VHDL International Users Forum*, Fall 1995, pp. 1.7-1.13.

[10] W. W. Dungan, R. H. Klenke, J. H. Aylor, "A 'Watch-and-React' Interface for Hybrid Modeling," CSIS Technical Report 960531.0, University of Virginia, May 31, 1996.

[11] Geist, G. A., V. S. Sunderam, "Network-Based Concurrent Computing on the PVM System," *Concurrency: Practice & Experience*, Vol. 4, No. 4, pp. 293-311, June 1992.

[12] D. Delany. Improving Ethernet LAN Performance with Ethernet Switching. PlainTree Systems, http://www.nstn.ca/plaintree/wpaper.html 1995.

**Appeared in the Proceedings of the IASTED International Conference on Modeling and Simulation, 1997, pp. 429-438**

# INTEGRATED PERFORMANCE AND DEPENDABILITY ANALYSIS USING THE ADVANCED DESIGN ENVIRONMENT PROTOTYPE TOOL ADEPT

Ramesh Rao, Arshad Rahman, Barry W. Johnson
Department of Electrical Engineering, University of Virginia
Charlottesville, VA 22903

## Abstract

The Advanced Design Environment Prototype Tool (ADEPT) is an evolving integrated design environment which supports both performance and dependability analysis. ADEPT models are constructed using a collection of predefined library elements, called ADEPT modules. Each ADEPT module has an unambiguous mathematical definition in the form of a Colored Petri Net (CPN) and a corresponding VHDL description. As a result, both simulation-based and analytical approaches for analysis can be employed. The focus of this paper is on dependability modeling and analysis using ADEPT. We present the simulation based approach to dependability analysis using ADEPT and an approach to integrating ADEPT and the Reliability Estimation System Testbed (REST) engine developed at NASA. We also present analytical techniques to extract the dependability characteristics of a system from the CPN definitions of the modules, in order to generate alternate models such as Markov models and fault trees.

## 1. Introduction

There exists a need for an integrated design environment that permits linking of the design phases from initial concept to the final physical implementation. An integrated design environment is important for several reasons. First, analysis capabilities for the early phases of the design process are greatly needed. Design alternatives, including hardware/ software trade-offs, may be more effectively evaluated with respect to multiple metrics, such as dependability and performance, in an integrated environment. Second, an integrated environment will allow concurrent and cooperative development of both hardware and software. Finally, true stepwise refinement in design optimizes the overall process and allows design verification techniques to be applied throughout the design process.

From the point of view of dependable system design, existing methodologies do not support the full range of system-level design and evaluation capabilities that are essential for the achievement of fault tolerance in critical applications. One of the major drawbacks of existing design methodologies is the need to switch between different environments and models during the different phases of a design and while performing different types of analyses.

The need to deal with several models of the same system during the design process results in three major problems: 1) any change in the design has to be correctly reflected across all the models of the system, 2) there is no way of ensuring that all the different models correspond to the same system, and 3) users must be familiar with several modeling languages and tools. Further, analysis of design alternatives is difficult and is likely to be limited by time constraints.

ADEPT overcomes the above mentioned drawbacks by providing an integrated environment based on a single modeling language and mathematical foundation. This unified approach has several significant advantages. First, a common modeling language and simulation environment that spans numerous design phases is much easier to use, encouraging more design analysis and consequently better designs. The common modeling language and simulation environment decrease the need for translators and multiple environments, reducing inconsistencies and the probability of errors in translation. Finally, the existence of a mathematical foundation provides an environment for complex system analysis using analytical approaches.

Simulators for hardware description languages accurately and conveniently represent the physical implementation of digital systems at the circuit, logic, register-transfer, and algorithmic levels. By adding system level modeling capability based on extended Petri Nets and queuing models as a mathematical foundation to the hardware description language, a single design environment can be used from concept to implementation. Such a methodology enables mixed simulation of both high level models and low level models due to the use of a common modeling language[139].

ADEPT is based upon such a methodology and uses the VHSIC Hardware Description Language (VHDL) for verification through simulation and colored Petri Nets for analytical approaches to analysis. A library of predefined elements (called ADEPT modules) has been developed from which systems can be constructed.

---

The authors would like to acknowledge the support provided by the Advanced Research Projects Agency under contract number F33615-93-C-1313, the Semiconductor Research Corporation under contract number 93-DJ-152, the International Business Machines Corporation under contract number WM-226181, NASA-Langley Research Center under contract numbers NCC1-173 and NGT-50578,, and Union Switch and Signal under contract number A-91846-24.

**Appeared in the Proceedings of the AIAA Computing in Aerospace Conference, 1995, pp. 285-300**

Performance and reliability models can be developed by interconnecting a collection of the ADEPT modules. ADEPT provides a graphical interface to aid in the actual construction of these models. Using ADEPT, the designer can avoid interaction with any VHDL code or the underlying Petri Net description.

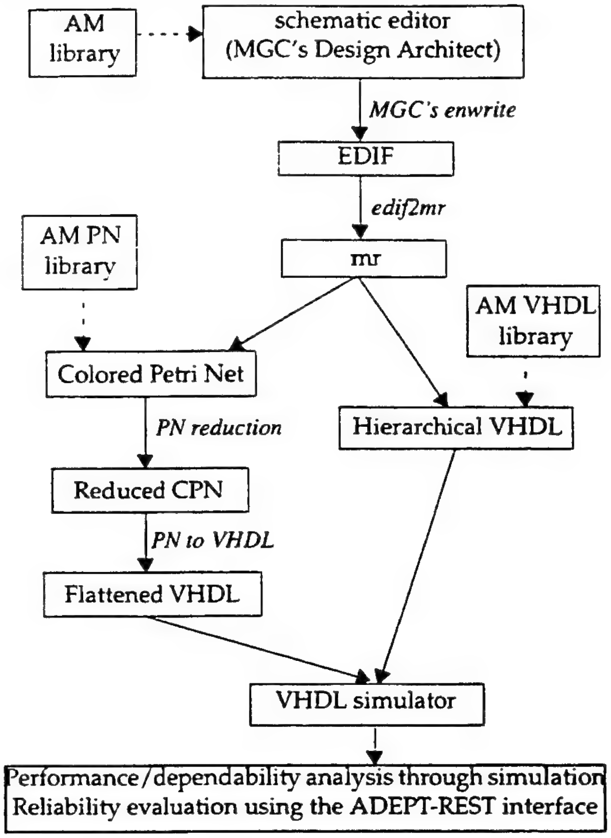
The remaining portions of this paper are organized as follows. Section 2 provides a brief overview of the ADEPT environment. Section 3 introduces the CPNs used in ADEPT. Section 4 discusses dependability modeling and analysis using ADEPT. Finally, a summary is provided in section 5.

## 2. ADEPT OVERVIEW

This section provides an overview of the ADEPT environment. A more detailed overview may be found in [144,163]. There are two Versions of ADEPT. Version 1 is currently available. Version 2 is an "alpha" version. ADEPT is available on a SPARC™ platform and uses Mentor Graphics' Design Architect (DA) [145] as the front end schematic capture system. DA is used to graphically construct the system model from a library of ADEPT module symbols. Both Versions automatically produce 1076 VHDL code [146]. Version 1 generates "flattened" VHDL code, that is, containing no hierarchy. Facilities and programs to collect and analyze the simulation results are provided as part of the ADEPT system.

One disadvantage of Version 1 is that the user is tied to a particular schematic capture system. Version 2 provides front end independence by utilizing EDIF as an intermediate hierarchical design format. Thus, any schematic capture system that can generate EDIF can be used as a front end to Version 2. Also, the capability of generating hierarchical VHDL code is provided. Finally, Version 2 supports model reduction using Petri Net reduction algorithms developed specifically for system models constructed from modules. These features are depicted in Figure 14. Although not explicitly shown in Figure 14, the CPN representation can also be used to perform analytical reliability analysis. This path (currently being implemented) allows the CPN description of the system model to be extracted, reduced and mapped to reliability models such as Markov models and fault trees. These techniques are presented in more detail in the section on dependability analysis using ADEPT.

Currently, the *enwrite* program within DA is invoked to generate an EDIF file. The resulting EDIF file is then used as input to an ADEPT program called *edif2mr* which generates a description of the system model in an internal ADEPT format called *mr*. Once a system model constructed out of ADEPT modules is translated into the internal *mr* representation, two paths exist for performance analysis. The user can simulate the hierarchical VHDL model generated. Alternatively, one can use the VHDL model generated from the reduced CPN model. The CPN reduction in the CPN to VHDL path in Figure 14 is used to simplify the PN model in the hope of reducing the simulation time.



AM = ADEPT Module

MGC = Mentor Graphics Corporation

**Figure 14.: ADEPT Version 2 (from [152])**

One of the key features of ADEPT is that the user interacts with a single model of the system, the ADEPT model. All simulation and analytical based studies of the system are performed on models that are automatically derived from the ADEPT model using provably correct transformations/mappings. This ensures that any change in the system design is correctly reflected in all models used in the analysis of the system. ADEPT has been designed to support several aspects of system design including integrated performance/dependability modeling [149,150,155,157], model reduction [152,153], hybrid modeling [159], hardware/software codesign [162], and the integration of operational specifications with performance models [160].

## 2.1 The ADEPT Modules

In the ADEPT environment, a system model is constructed by interconnecting a collection of *ADEPT modules*. The modules model the information flow, both data and control, through a system. The modules communicate by exchanging *tokens*, which represent the presence of information, using a uniform, well defined handshaking protocol [141]. Higher level modules can be constructed from the basic set of ADEPT modules. In addition, custom modules can be incorporated into a system model as long as the handshaking protocol is adhered to.

A token is implemented as a VHDL record structure. In the token, the two most important fields are the STATUS field and the COLOR field. The STATUS field is used to implement the token passing mechanism, that is, the "handshaking" between the ADEPT modules. The COLOR field, which is itself a record structure, is used to hold user-specified information. Modules are provided which can manipulate the information in the COLOR field.

An example of an ADEPT module is the *Wye*, shown with its underlying colored Petri Net representation in Figure 15. This module models a "fork" construct. When a token arrives at In1, tokens are placed simultaneously at Out1 and Out2. The input token is not acknowledged (consumed) until both output tokens have been acknowledged.

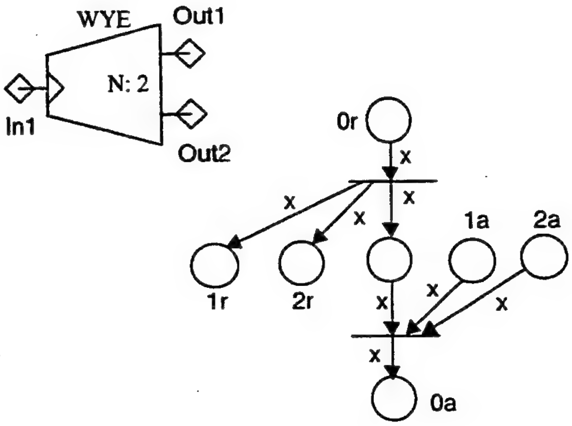


Figure 15. Wye Module and CPN Representation

In the Petri Net of Figure 15, the "r" and "a" labels correspond to "ready" and "acknowledge", respectively. The ready and acknowledge places emulate the handshaking between modules. When a token arrives at the place labeled "0r", the top transition is enabled, and a token is placed in the "1r", "2r", and center places. The first two places correspond to a token being placed on the module outputs (Out1 and Out2). Once the output tokens are acknowledged (corresponding to tokens arriving at the "1a" and "2a" places), the lower transition is enabled, and a token is placed in "0a" (corresponding to the input token being acknowledged). The module is then ready for the next input token. Other modules are modeled similarly. The complete CPN descriptions of each of the ADEPT modules can be found in [142].

The entire set of ADEPT modules is divided into six categories: control modules, color modules, delay modules, fault modules, miscellaneous parts modules, and hybrid modules. The control modules, except the *Switch*, *Queue*, and logical modules, have been adapted from Dennis [143]. The *Wye* module described above is an example of a control module. ADEPT modules in the color and delay categories enable the manipulation of the token color and model temporal aspects of a system, respectively. The fault modules are used to model the presence of faults and errors in a system model. The miscellaneous parts category contains modules that are used for data collection with the ADEPT system. The last category contains modules which aid in the construction of hybrid models. A more detailed description of the entire ADEPT module set can be found in [144].

## 3. Colored Petri Nets

This section informally introduces the CPNs used in ADEPT. A formal treatment of CPNs may be found in [140]. The CPNs used here consist of three parts: net structure, declarations, and net inscriptions. CPNs have the same structure as ordinary Petri Nets. A CPN distinguishes itself from other PNs by allowing its tokens to have complex data types called colors. The declarations of these data types form the declaration part of the net. The places, the transitions, and the arcs of CPNs have inscriptions which determine the behavior of the net.

A place can have three kinds of inscriptions: (1) *Name* merely distinguishes a place from other nodes; (2) *Color set* decides what kinds of tokens can reside in that place; (3) *Initial marking*: a multiset of tokens belonging to the place's color set. A multiset n'a+m'b means there are n copies of element a and m copies of element b in the set.

A transition can have two kinds of inscriptions *Name* and *Guard expression*.The guard expression of a transition (which evaluates to true or false) must be true before the transition is enabled. The guard expression which always evaluates to true is omitted. In the guard expressions, operators ==,!=, &&, ||, ^^ mean equal to, not equal to, and, or, and xor respectively.

An arc has only one inscription and it is called the *arc*

**Appeared in the Proceedings of the AIAA Computing in Aerospace Conference, 1995, pp. 285-300**

*expression*. The arc expression evaluates to a multiset of elements belonging to the color set of the place on one of its ends.

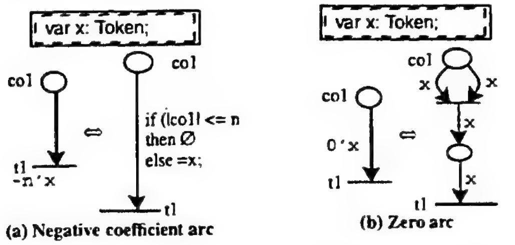
*Dynamic Behavior of Colored Petri Nets*

All the variables that are associated with a transition are bound to colors of their respective types. A transition T is enabled by a binding b, if each of its input places has at least those tokens that the corresponding arc expression evaluates to under the binding b, and the guard expression associated with the transition evaluates to true under the binding b. Such an enabled transition T is said to fire when it removes those tokens evaluated by the arc expressions from each of the corresponding input places. Time is introduced into the CPN through a special guard expression called the wait expression. The timed transitions using wait expressions do not appear in Jensen's work[140]. If the guard expression is a wait expression, the transition will remove the tokens from the input places and wait for an amount of time stated in the wait expression before placing tokens on the output. When a transition fires, it will add tokens to each of its output place according to its corresponding output arc expression value under the current binding of the transition.

*Hierarchy in Colored Petri Nets*

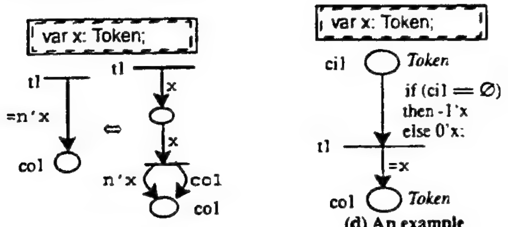
Hierarchy in CPNs helps one abstract away the commonly used sub-nets into nets of their own that can be used by several CPNs. A place, a transition, and an arc that can be expanded into a sub-net are called a super place, a super transition, and a super arc respectively.

Figure 16 illustrates the super arcs that are used in modeling the ADEPT modules. A place name within an expression will denote the entire multiset of tokens currently in that place.



(a) Negative coefficient arc

(b) Zero arc



(c) Equal to arc

(d) An example

Figure 16. Super arcs.

The negative coefficient in a multiset expression m'a-n'b in the input arcs means that the corresponding place must have at least m tokens of color a and at most n tokens of color b for the transition to be enabled. When the transition fires, it will remove m tokens of color a and none of color b. A zero coefficient in an input arc expression 0'a means that its input place must have at least one token of color a for the transition to be enabled. When the transition fires it will not remove any token of color a from the corresponding input place. Finally, an "=" symbol in front of an output arc expression means that when the transition fires it replaces the contents of the corresponding output place with the multiset of tokens evaluated by the arc expression.

The negative, the zero, and the equal-to arc expressions merely simplify the static structure of the CPN; they do not extend it. Figure 16d illustrates the use of these three arc expressions.

## 4. Dependability Modeling using ADEPT

One of the goals of the UVa. design methodology is to simplify the modeling and analysis of fault-tolerant systems. The aim is to allow design engineers as opposed to reliability engineers to model and analyze fault tolerant systems during the early stages of the design process. In our experience, we have found that most system designers do not feel comfortable with nor do they fully understand Markov models, fault trees, and other common reliability evaluation techniques. These factors often postpone reliability analysis until the later stages of the design. The UVa. design environment enables the system designers to model and analyze fault-tolerant systems within a data/control flow paradigm. System designers are familiar with such models since most systems level performance and functional models are also based on the data/control flow paradigm.

A subset of the ADEPT modules, called the Fault modules, are used to model fault injection, fault/error detection, error correction, and repair processes. Faults and errors are represented on a specific color field referred to as the *fault* field: a value of "true" indicates that a "fault" or "error" is present, and a value of "false" indicates that no "fault" or "error" is present. The Simple Fault or Fault-Injection module are used to inject faults into a system model, according to a user-selected failure density function. Although the exact distributions may not be known during the initial phases of a new design, a designer can choose distributions from experimental or field-test data from similar systems that have been used in the past, or can vary the parameters to perform a sensitivity analysis.

A typical use of the fault modules is illustrated in Figure 17. Here, the fault modules have been used to model a processor with a self-checking capability. The processor model itself is built using ADEPT modules. The fault injection module on its output models the failure of the processor, and the fault detection module models the self-checking process. Other fault-tolerant characteristics such as reconfiguration and repair are also modeled in a similar fashion.

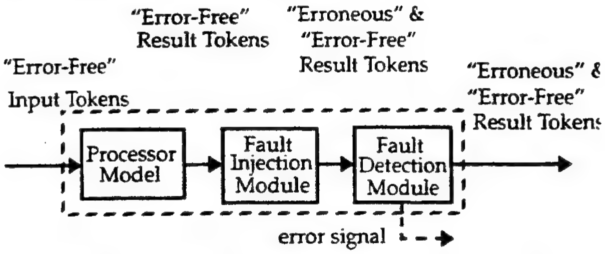

"Error-Free" Result Tokens
"Erroneous" & "Error-Free" Result Tokens
"Error-Free" Input Tokens
"Erroneous" & "Error-Free" Result Tokens
Processor Model
Fault Injection Module
Fault Detection Module
error signal

**Figure 17. Processor with Self-Checking Capability**

Thus, the fault-tolerant characteristics are embedded into the same model used to model the functional and behavioral characteristics of a system. Figure 18 shows the dependability analysis schemes supported by ADEPT. The starting point for each solution is an ADEPT module representation of the system. The ADEPT model is automatically mapped into CPN and VHDL models which are equivalent by construction. Either model (CPN or VHDL) can be simulated to evaluate several performance and dependability characteristics. Using the ADEPT-REST interface, the designer can use the REST engine to obtain lower and upper bounds on the reliability of the system. Further, several analytical models of the system such as Fault-trees and Markov models can be automatically derived. The following subsections present an example to illustrate the modeling of a fault-tolerant system using ADEPT followed by a discussion of the three techniques mentioned above.

## 4.1 Modeling Example

This section briefly describes the ADEPT model of a TMR (Triple-Modular Redundancy) system. Triple Modular Redundancy is the most common form of passive hardware redundancy in fault-tolerant systems. In a TMR system the outputs of three components is voted upon to produce a final output. Such a system can tolerate the failure of any one of the three components, since the two fault-free components will "mask" the erroneous output of the faulty unit. However, when any two or more of the components fail, the system will fail since the majority voter cannot determine the "correct" system output.

Figure 19 shows a high level ADEPT model of a TMR system. The model has five hierarchical components: the System Input Block, the three computers, and the voter. The System Input Block drives the model and produces tokens as quickly as they can be accepted by the system. The Wye module passes a copy of the token to each of the three computers. The computers "process" the token by generating a delay based on information contained in one of the color fields of the token. The computers also provide a *status* output which indicates to the voter their operational status ("working" or "failed"). The voter produces a token on its output which is then consumed by a Sink module,

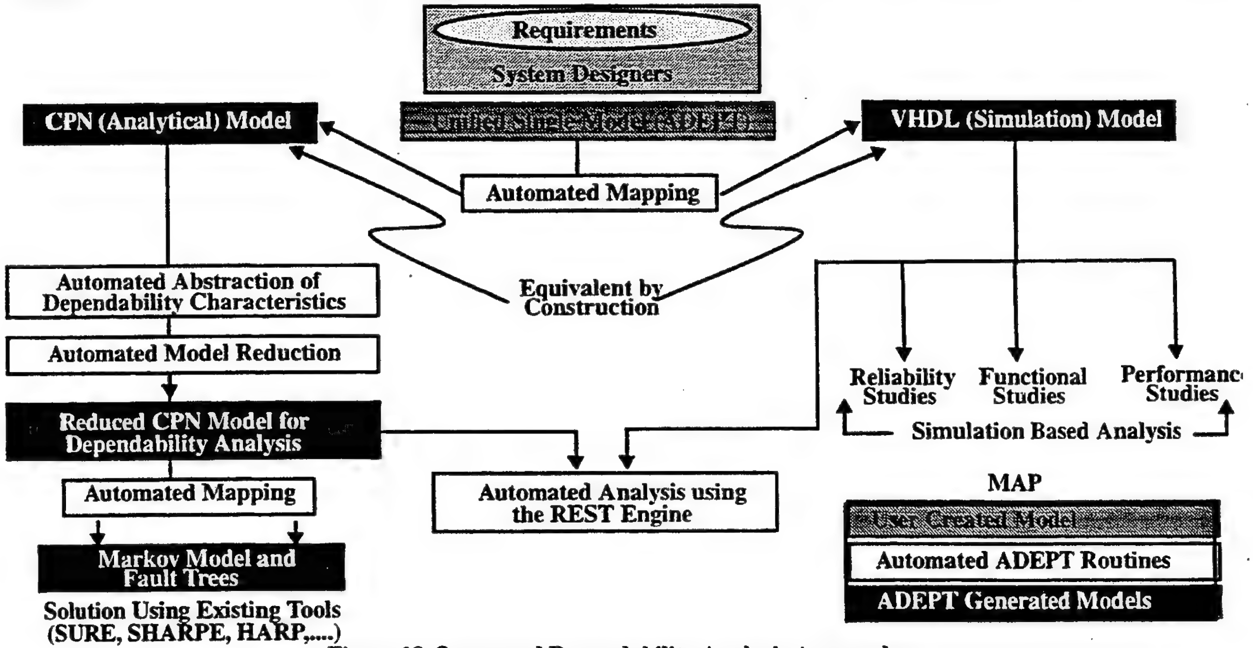

Requirements
System Designers
CPN (Analytical) Model
Unified Single Model (ADEPT)
VHDL (Simulation) Model
Automated Mapping
Equivalent by Construction
Automated Abstraction of Dependability Characteristics
Automated Model Reduction
Reduced CPN Model for Dependability Analysis
Automated Mapping
Markov Model and Fault Trees
Solution Using Existing Tools (SURE, SHARPE, HARP,....)
Automated Analysis using the REST Engine
Reliability Studies
Functional Studies
Performance Studies
Simulation Based Analysis
MAP
User Created Model
Automated ADEPT Routines
ADEPT Generated Models

**Figure 18. Supported Dependability Analysis Approaches**

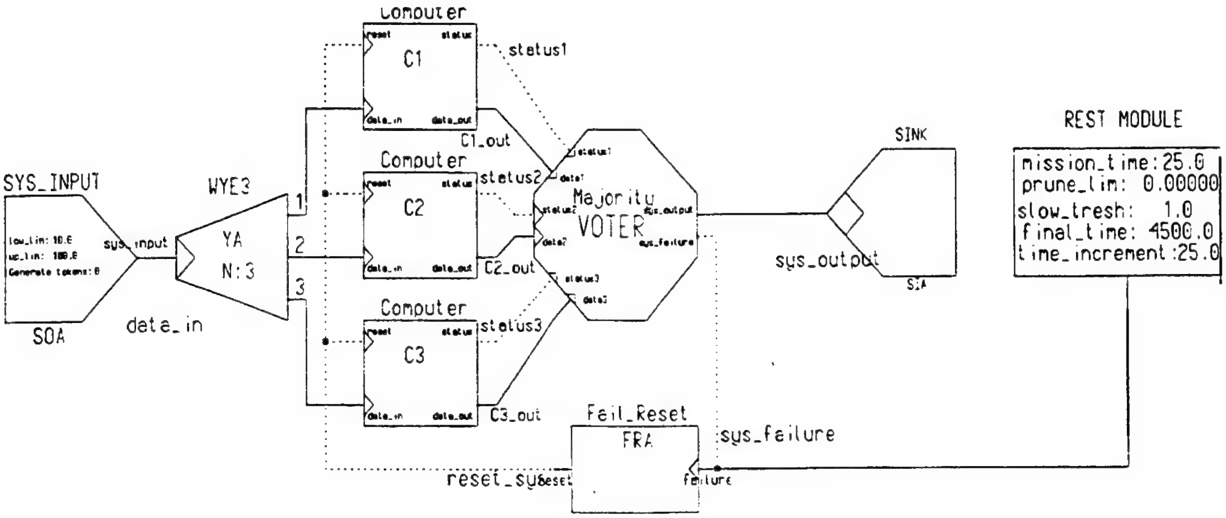


Figure 19. TMR System Model

representing some external data sink. The voter also provides a *sys_failure* output which becomes active whenever a system failure occurs.

The computer is modeled in a fashion similar to that shown in Figure 17. The *status (error signal)* output or the fault field of the computer output can be used to track component failures during the simulation. The implementation of the voter is shown in Figure 20. Since the system will fail when any two or more of the three computers have failed, the K-of-M module at the top of the figure causes the *sys_failure* output to go active when this condition arises. The Junction module ensures that a token is placed on the on the output of the voter only after all three inputs have arrived. The Set Color, Constant, and Set Fault modules together color the relevant field to "true" if there is a system failure, and "false" otherwise.

The REST module is connected to the *sys_failure* signal and uses the information on this signal to inform the REST engine if a loaded state results in a system failure. In the REST analysis mode, the state of the fault modules in the model are manipulated and controlled, through the STYX[148] interface, by the REST engine. The System Reset Block is used in the simulation based approaches to reliability and/or performance evaluation. ADEPT can automatically generate the CPN model of this system and extract reliability models such as markov models.

## 4.2 Analytical Approaches

This section demonstrates the analytical approaches to dependability evaluation in ADEPT. As illustrated in Figure 18, the CPN model of a system is first abstracted in a simpler CPN, using a set of transformations that eliminate information about the system that is not needed for reliability analysis. The state space of the abstracted CPN is then reduced, using a set of application specific CPN reduction rules. This reduced

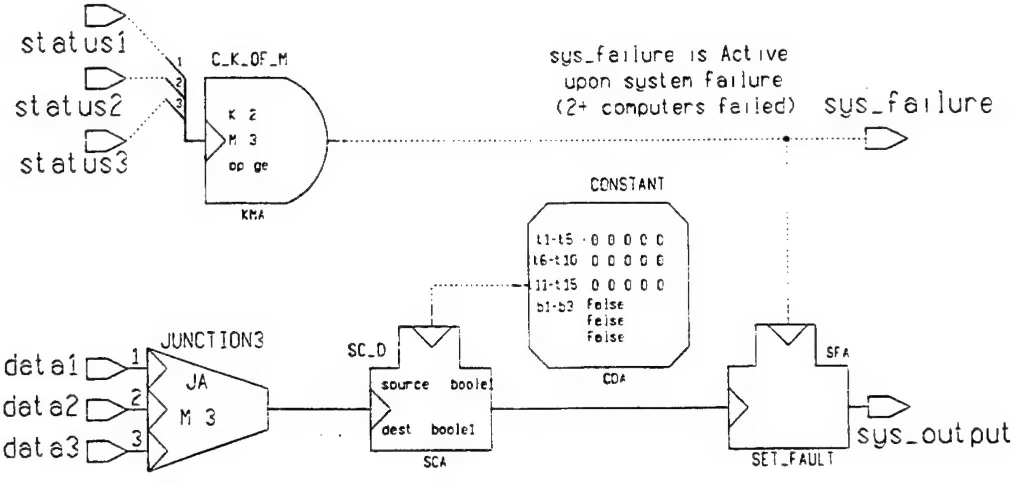


Figure 20. Voter Model

CPN may then be transformed into alternate models such as Markov models for the evaluation of system reliability.

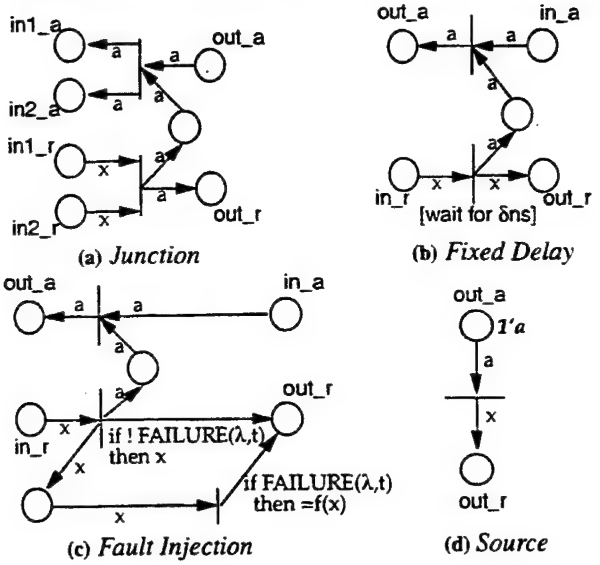
### 4.2.1 ADEPT module Descriptions

This section presents sample ADEPT modules used to illustrate the process of generating abstracted CPNs and Markov models.

The CPN definitions of four example modules are shown in Figure 21. The *Source* module initially generates a token on o1_r and thereafter, generates a token on o1_r every time an acknowledge token arrives on o1_a. An operation dependent on the availability of two independent pieces of information is modeled using the *Junction* module. A token is placed on the output of a *Junction* module only when a token is present at both its inputs. The input tokens are acknowledged only after the token at the output has been acknowledged. The token output by the *Junction* module has its *color* and *fault* fields reset to the default initial value since the propagation of the value on these fields is application dependent. The error and color propagation characteristics of the *Junction* module must be explicitly specified by the user.

The *Fault_Injection* module places a token on its output as soon as a token arrives on its input. Associated with the module is a failure rate, based on which the *fault* field of the output token is set to either true or false. The *Fixed Delay* module produces a user specified delay between the time a token arrives at its input and is transferred to its output.

```
var x,y: Token;
const a: Default_initial_Token;
const δ: integer;
const λ: real;
Token f (Token x) {
      x.fault = TRUE;
      return x;
}
Boolean FAILURE (Token x, Real λ, Time t) {
      \*implementation dependent *\
}
```



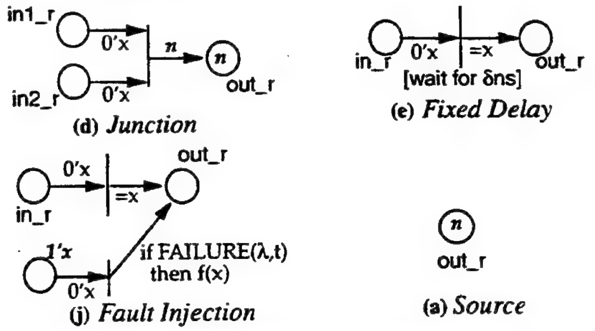
(a) *Junction* (b) *Fixed Delay*

(c) *Fault Injection* (d) *Source*

**Figure 21. Sample CPN definitions**

*Abstracted CPN Descriptions*

From a reliability point of view, it is the propagation of erroneous tokens through the system that is of interest. Thus, those arcs in the CPN definitions of the ADEPT modules which do not propagate erroneous tokens may be removed. By definition, the token output by the Source module always has its *fault* field set to false. Hence, the reduced CPN description of the *Source* module is an empty input place. The abstracted CPN model of the system is driven by the *Fault* modules that introduce erroneous tokens into the system. Thus, the default arc expressions on input arcs to transitions may be replaced by *0'x*, where *x* is a variable of type *Token*, and the default arc expressions on output arcs from transitions may be replaced by *=x*. Further, we assume that the data processing rates are much greater than the failure rates and the delays introduced by the delay modules can be ignored. Since we are interested in the probability of an erroneous token reaching the system output, the acknowledge paths in the CPN definitions of modules may be eliminated. The abstracted CPN definitions for the ADEPT modules presented above are shown in Figure 22.



(d) *Junction* (e) *Fixed Delay*

(j) *Fault Injection* (a) *Source*

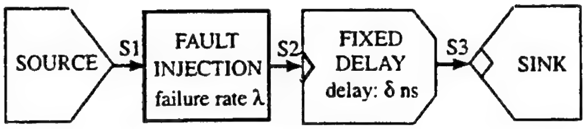
**Figure 22. Abstracted CPN Descriptions**

From a reliability standpoint, the equivalence between the two CPN representations may be established in the following fashion. Consider the ADEPT model of a simple series system shown in Figure 23a. The *Source* module places a token at the input of the *Fault_Injection* module. The *Fault_Injection* module determines if a failure occurs at that point in the system based on a user specified failure rate and current simulation time. Once a failure occurs, the *Fault_Injection* module starts setting the *fault* field of the tokens passing through it to true, indicating that the data at that point in the system is erroneous as a result of the failure. If no failure occurs the *Fault_Injection* module leaves the *fault* field of the token untouched. The flow of tokens through the *Fault_Injection* module occurs in zero simulation time. The *Delay* module models the processing delay of the system.

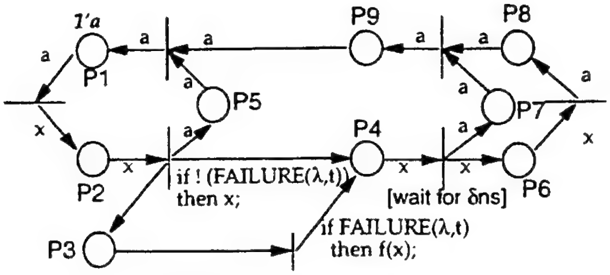
Figure 23b shows the CPN model obtained using the complete CPN definitions of the building blocks. The set of possible markings for the CPN model is shown in Figure 23c. In Figure 23c $a_e$ is used to represent a token with its fault field set to true. Note that initially the system keeps cycling through states m1 - m6 until the failure occurs. Once the failure occurs, the system makes a transition to, and keeps cycling through, a second set of states m7-m13. Since the failure transition rate λ is much less than the data processing rates and the failure transition is of interest, the states m1-m6 can be aggregated into a single state and the states m7-m13 can be aggregated into a single state. The resulting Markov model for the system is shown in Figure 23d. This Markov model for the system is what is expected for a simple series system since such a system has only two states from the reliability standpoint, operational (m1-m6) and failed (m7-m13), and the failure probability is driven by the single failure rate λ.

Figure 23e shows the CPN mapping of the system obtained using the reduced CPN descriptions of the building blocks. The *SOURCE* module maps into an empty input place and does not affect any state changes of the system. Thus, the place P1 can be deleted. Since the transition from place P3 to P4 is instantaneous the two are combined into a single place. The resulting CPN is shown in Figure 23f and has only two markings, one with a token in place P2 and the other with a token in both places. The transition between these markings is governed by the failure rate λ. The equivalent Markov model for Figure 23f is identical to the one obtained using the Complete CPN definitions of the building blocks. The reduced CPN descriptions of all the building blocks can be shown to be equivalent to their corresponding complete CPN definitions in a similar fashion. For the formal definitions and proofs of the aggregation techniques presented here the reader is referred to [170,171,172].

The abstracted CPN description of the *Source* module is an empty input place. Since the reduction process presented here is hierarchical in nature, the abstracted CPN description of the *Source* module is marked with a null token *n* to distinguish it from the input places of submodels in the hierarchy. In order to generate Markov models from ADEPT model the user needs to identify those system outputs that contribute to system failure. In the mapping process, the places corresponding to these

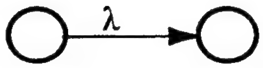


(a) A Simple Series System

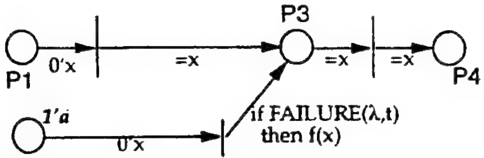


(b) Series System CPN Model

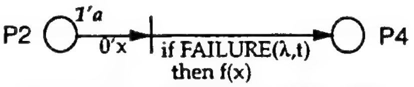
(c) Possible Markings for System CPN Model



(d) Resulting Markov Model



(e) Series System CPN Model



(f) Series System Reduced CPN Model

Figure 23. CPN-Abstracted CPN equivalence

outputs are tagged and these places are not removed in the reduction process. Output places that do not contribute to system failure are marked with a null *n* token and can be removed in the reduction process.

### 4.2.2 CPN Reduction Rules

This section presents a couple of sample reduction rules that will be used to illustrate the reduction process. The reduction rules are based on the error propagation characteristics of the ADEPT model and are specific to this methodology. The rules are node elimination rules that focus on reducing the state space of the model. Figure 21 illustrates two such rules. In Figure 21 the default arc expression on input arcs to transitions is 0'x where x is a variable of type token. The default arc expression on output arcs from transitions is =x.

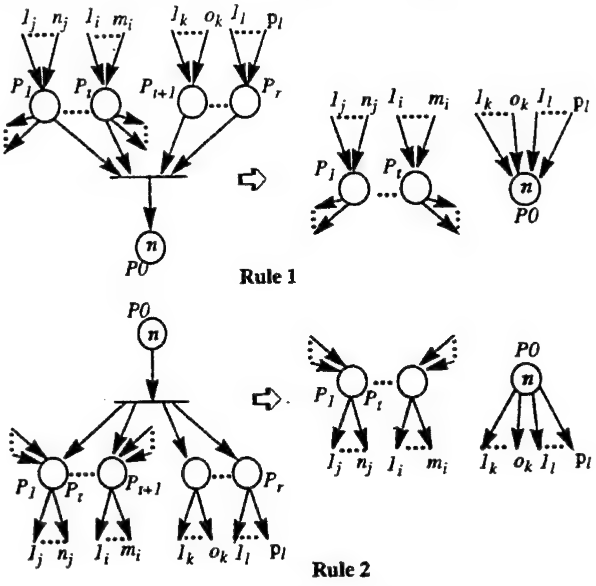

$l_j$ $n_j$ $l_i$ $m_i$ $l_k$ $o_k$ $l_l$ $p_l$
$P_1$ $P_t$ $P_{t+1}$ $P_r$
n
P0
Rule 1
Rule 2

**Figure 24. Sample Reduction Rules**

### 4.2.3 Example

This section presents an overview of the reduction process applied to a TMR with a spare. In this example the ADEPT module model is first mapped onto the corresponding CPN model using the abstracted CPN definitions of the ADEPT modules. The CPN model is reduced using the reduction rules and then converted to a Markov model using techniques similar to those described in [173]. The faults are assumed to be permanent, non-simultaneous faults.

An overview of the TMR with a spare is shown in Figure 25. In this example, it is assumed that there is some form of fault detection in P3 that disconnects P3 when it fails and brings P4 on-line. It is assumed that the coverage factor is 1, which is not a limitation of this approach.

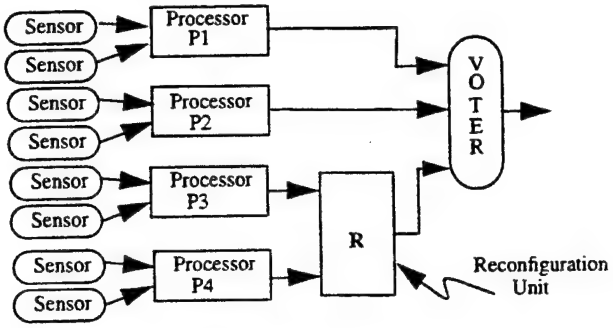
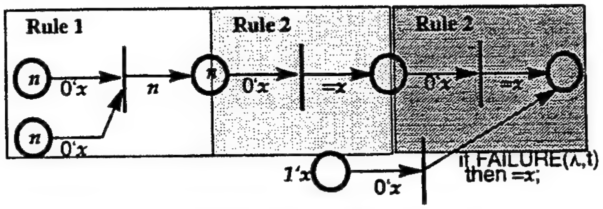
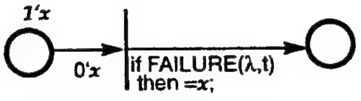


**Figure 25. TMR with A Spare**

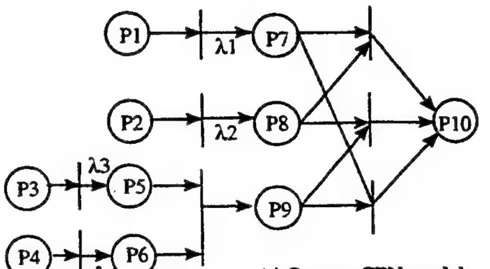
The processor shown in Figure 25 is modeled in a fashion similar to the processor shown in Figure 17. The CPN model of the processor (Figure 26a) is obtained by replacing each ADEPT module by its corresponding reduced CPN definition. Rules used to reduce the CPN in Figure 26a to the CPN in Figure 26b are also illustrated. The remaining components of the system are reduced in a similar fashion and are combined to obtain the reduced CPN representation shown in Figure 26c. The corresponding Markov model is also shown.



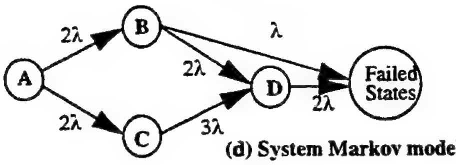
**(a) Corresponding CPN Model**



**(b) Reduced CPN Model**



**(c) System CPN model**



**(d) System Markov model**

**Figure 26. Processor Model Reduction**

### 4.3 Simulation based Approaches

In the simulation based approach, the underlying VHDL description of the system is used for reliability analysis. Simulation provides a convenient mechanism for integrating performance and reliability evaluation. Since the *functional model* can be used for reliability evaluation the fault/error handling, reconfiguration, and recovery processes can be explicitly modeled exactly the way they will be implemented in the final design. No limitations are placed on the types of probability distributions that can be selected to govern the fault/ error characteristics of a system. Thus, the system's actual response to failures will be more accurately represented and more precise evaluations can be carried out. Reliability and performance can be studied simultaneously from the single model, so the interaction between the two measures can be completely observed. This integration is especially important for real-time systems, since the performance and reliability depend so heavily upon each other.

The basic idea behind this relatively simple approach[150] is to run a simulation until the system fails, and record the time of failure. To improve simulation efficiency, multiple failure cycles can be simulated in a single, long simulation run using the concept of regenerative simulation. From the collection of failure times, the failure density function, $f(t)$ can be derived, from which the system reliability $R(t)$ can be estimated[164]. The failure density function $f(t)$ describes the probability for system failure in $dt$ about $t$ per unit time [165]. Reliability $R(t)$ is defined as the probability that the system continues to operate correctly throughout the interval $[t_0, t]$, given that it was operating correctly at time $t_0$[166]. The two functions are related by the equation

$$R(t) = \int_{t}^{\infty} f(\tau)d\tau$$

A simulation-based approach using this equation would be to: (1) run $N$ failure cycles and collect the failure times, (2) form a histogram of the collection of time-to-failures, (3) construct $f(t)$ by dividing each histogram entry by $N$, and (4) estimate $R(t)$ by adding up the "area" under $f(t)$ from $t$ to $\infty$.

An equivalent, and more convenient approach is to use an indicator function. Here, we can estimate the reliability as: $R(t)=S(N)/N$, where $S(N)$ is the observed number of "no failure" trials in $N$ trials of duration $t$. The approach using this method would be to: (1) run $N$ failure cycles and collect the failure times, (2) form a histogram of the collection of time-to-failures, and (3) estimate $R(t)$ by summing up the histogram entries (which are the number of failures in each time block) from $t$ to $\infty$ and dividing the result by $N$. Since the histogram orders the failure times, $S(N)$ for a trial of length $t$ is just the sum of the histogram from $t$ to $\infty$ (here, $\infty$ is really just the largest observed failure time, since we know all histogram entries are zero after this time). All histogram entries less than $t$ represent failures in a cycle of length $t$. In this fashion, $R(t)$ can be found for all integer values of $t$ from a single set of simulations, rather than running multiple sets of simulations for different lengths of $t$, as the basic $\{S(N)/N\}$ approach would suggest. The number of trials for each t is the same (N), since we use the failure density function histogram to determine "failure" and "no failure" trials for each value of t, from which reliability is calculated. If the histogram entries are set at unit widths, this approach gives better accuracy.

Conventional simulation techniques are impractical for ultra-reliable systems due to the extremely low failure rates involved, which could require billions of long simulation runs to produce accurate results. It has been shown[167] that for failure rates on the order of $10^{-9}$, more than 1 trillion trials are required for a 95 percent confidence level on reliability using standard simulation techniques. However, recent advances in *variance-reduction techniques* have been shown to reduce the number and length of simulation trials by orders of magnitude over conventional techniques[168]. The primary variance-reduction technique used for analysis of highly-reliable systems is known as *Importance Sampling*, which has been shown to reduce the number and length of simulations by orders of magnitude. Importance sampling is currently being incorporated into ADEPT as an alternative evaluation method to that described above.

To support this simple approach ADEPT provides a post simulation tool which automatically calculates the reliability $R(t)$ for specified values of $t$ using a file containing the system failure times collected during simulation. These times must be collected by a Collector module connected to the *system_failure* signal. The simulation based approach has been used on a rich set of examples. The results of some of these examples are presented along with the results obtained using the ADEPT/REST solution method.

### 4.4 ADEPT/REST Solution method

This section describes the ADEPT-REST interface. The REST[151] engine, developed at NASA-Langley and the College of William and Mary, supports simulatable failure modes and effects analysis and automatically produces a Markov model of the system which can then be analyzed with a reliability Markov engine. REST provides, among other things, lower and upper bounds on the system unreliability.

The REST engine is interfaced to the ADEPT VHDL model. In order to obtain the REST solution using this approach, all a designer has to do is connect the ADEPT-REST module (an ADEPT module) to a signal that goes active in the event of a system failure. The ADEPT-REST module has generics that are used to pass the mission time, pruning level, and the threshold for slow transitions to the REST engine. Further, the user may obtain multiple solutions of a given model by specifying a range of mission times and an increment as generics to the REST module. All the other interaction between the ADEPT model and the REST engine is automated by the ADEPT-REST interface. Upon completion of the reliability evaluation, the ADEPT-REST interface automatically displays the results in the form of a graph.

An overview of this approach is shown in Figure 27. The ADEPT VHDL model communicates with the ADEPT-REST interface (written in "C") via the Vantage VHDL-C interface (STYX). The ADEPT-REST interface will work with any VHDL simulator that supports external language function calls. In order to use the ADEPT-REST interface with a VHDL simulator the user needs to modify a package (rest_interface.pkg) in which the function call to the ADEPT-REST interface is declared. The function has four parameters (two integer and two real) and linking this function to the ADEPT-REST interface depends on the simulator being used.

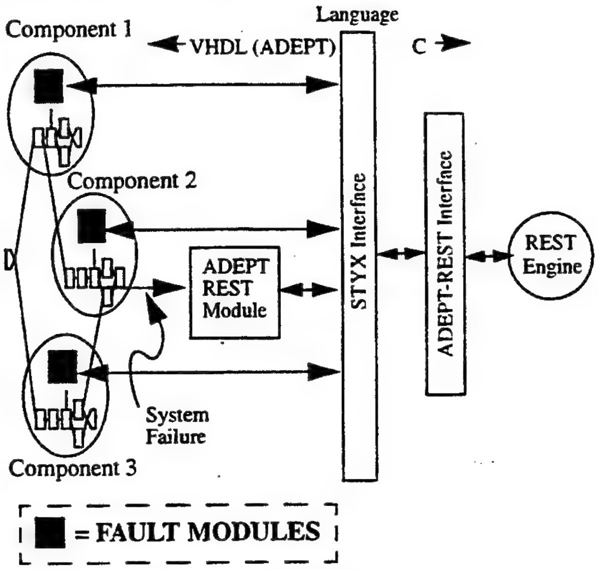


Figure 27. ADEPT-REST Interface

Each fault module that can potentially cause a change in the system state from a dependability point of view, constitutes one element of the system's global state vector. Examples of such modules include the Simple Fault module, the Fault Injection module, the Reconfiguration module, and the Repair module. Hence, a change in the local state of such fault modules causes a change in the global system state. An example of a local state change is the Simple Fault module going from an unfailed state to a failed state. In essence, each fault module is a transition that causes the system to go from one state to another.

In the approach presented here the fault modules wake up on odd simulation cycles and the ADEPT-REST module wakes up on even simulation cycles. The function call to the ADEPT-REST interface is used to pass the operation desired into and out of the ADEPT REST interface. The operation of the interface is presented in Table 1.

The interaction between the ADEPT model and the REST engine consists of the following basic events: 1) the REST engine requests each fault module to load, or set itself to, a specified state. For example, a simple fault module may be set to the failed or unfailed state. 2) the ADEPT-REST module informs the REST engine if the loaded state causes system failure. 3) if the loaded state does not cause system failure, the REST engine requests the model to provide all possible states it can transition to along with the associated transition rates. Based on this information the REST engine builds a semi-Markov model which it solves using the SURE[169] analysis program.

The analysis using the REST engine proceeds in the following fashion: during the initialization phase each fault module that can potentially cause a change in the system state writes its initial state into the global state vector. This initial global state vector along with the mission time, pruning level, and the slow transition treshold is passed to the REST engine through the ADEPT-REST interface (steps 0-4 in table 1). Next, The three basic steps mentioned in the previous paragraph are repeated for each possible state the system can take starting from the initial state provided to the REST engine (steps 5 and 6 in table 1). Upon completion of its analysis the REST engine reports the upper and lower bounds on the unreliability of the system along with the number of states pruned.

This approach to interfacing the ADEPT VHDL model and the REST engine requires that all data processing delays in the ADEPT model be set to zero. This is necessary since the system must settle to a steady state, from any loaded state, in zero simulation time. This can be achieved in two ways. One approach is to block the tokens at the output of all source modules and enumerate all the system failure conditions. An alternative approach is to use a top level generic to set all processing delays in the model to zero during the REST analysis.

## 4.5 Examples and Results

This section briefly describes the ADEPT models of a reconfigurable quad and the FTPP (Fault Tolerant

## Table 1: Operation of the ADEPT-REST interface

| Time | Fault Modules | ADEPT-REST Module | ADEPT-REST Interface |
|---|---|---|---|
| 0 ns | Initialize Variables | Initialize Variables | Initialize Variables |
| 1 ns | Get id from interface | | Update vector length and return as id. |
| 2 ns | | tell interface to allocate memory for state vector | allocate array of size vector length |
| 3 ns | pass id and initial state to interface | | use id as index into statevector array and load initial state |
| 4 ns | | pass pruning level, mission time, and slow transition treshold to interface. | initialize REST engine:<br>1) mode = pointer passing and exhaustive<br>2) capabilities = complete<br>3) send mission time etc. to REST engine<br>4) send begin signal and initial state<br>5) obtain current analysis mode (load state vector, test failure condition, test transitions, or terminate) from REST engine |
| **The above 4 steps complete the initialization phase.** | | | |
| 5 ns | send id and request analysis mode | | if current analysis mode = load state vector<br>then return analysis mode and new_state<br>else return analysis mode<br>end if |
| | if analysis mode =<br>load state vector<br>then state=new_state<br>end if<br><br>if analysis mode =<br>test transition<br>then<br>if state change possible<br>then send next state<br>and transition rate<br>end if<br>end if | | if current mode = test transition and<br>state change possible<br>then<br>allocate space for next state vector<br>next state vector = loaded state vector<br>next state vector[id] = next state<br>end if |
| 6 ns | | send system failure condition to interface | if analysis mode = test death state<br>then<br>send system condition to REST engine<br>end if<br><br>if analysis mode = test transition<br>then<br>send next state vectors and rates<br>end if |
| **Repeat steps 5 and 6 and alternate simulation times until terminate signal is received.** | | | |

**Appeared in the Proceedings of the AIAA Computing in Aerospace Conference, 1995, pp. 285-300**

Parallel Process). Several results obtained using the simulation and ADEPT-Rest solutions are also presented.

### 4.5.1 A Reconfigurable Quad

The Reconfigurable Quad serves as an important building block in many fault-tolerant systems. The system consists of four computers operating in parallel, with a reconfigurable majority voter to mask any errors that may occur. The system can reconfigure from a quad to a TMR and thus can operate with only two fault-free processors, given that it has reconfigured successfully. Upon the first processor failure, the system attempts to reconfigure to a triad. If a second processor fails during the reconfiguration of the first, the system will fail since a majority output cannot be determined. However, if the second failure occurs after the first processor has been successfully reconfigured, the system will still be operational. After three or more of the processors have failed, the system will fail since a majority output cannot be determined.

The ADEPT model for the Reconfigurable Quad system is shown in Figure 28. The model has the following hierarchical components: the System Input Block, the computers, Remove_Computer Block, and the voter. The System Input Block drives the model and produces tokens as quickly as they can be accepted by the system. When a processor fails, the reconfigured output of the associated Reconfigure module becomes active upon successful reconfiguration, shutting down the output of the processor in the Remove_Computer Block. The purpose of the Remove_Computer Block is to block the output of a failed computer after it has been successfully reconfigured. The Quad Voter provides a "sys_failure" output which becomes active whenever a system failure occurs.

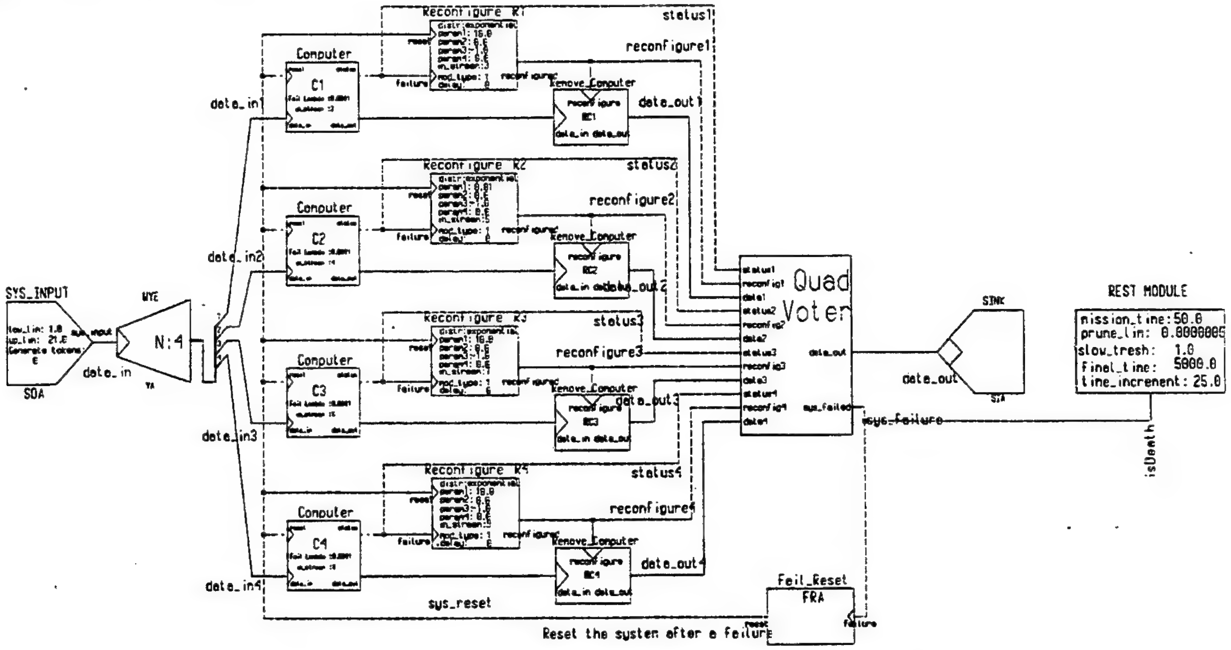


Figure 28. Reconfigurable Quad System ADEPT Model

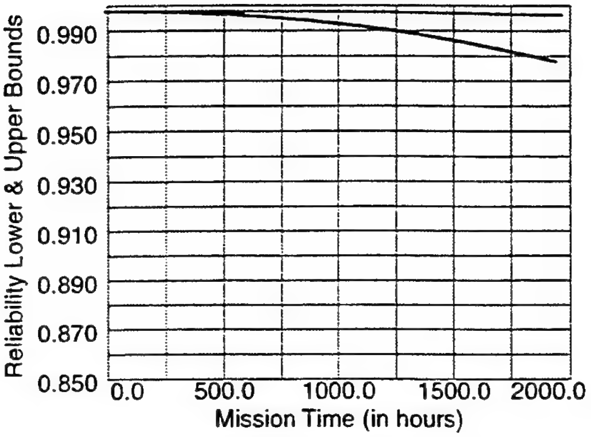
*Reconfigurable Quad Analysis Results*

This section presents some of the results obtained using the ADEPT-REST interface. In the results presented here the mission time was evaluated from 50 to 2000 hours in increments of 25 hours. The plot for an exponential failure rate of 0.00005 failures/hour is shown in Figure 29a. The reconfiguration rate was set at 3600 reconfigurations/hour. A comparison of the results obtained using the simulation based approach and the ADEPT-REST approach is shown in Figure 29b.

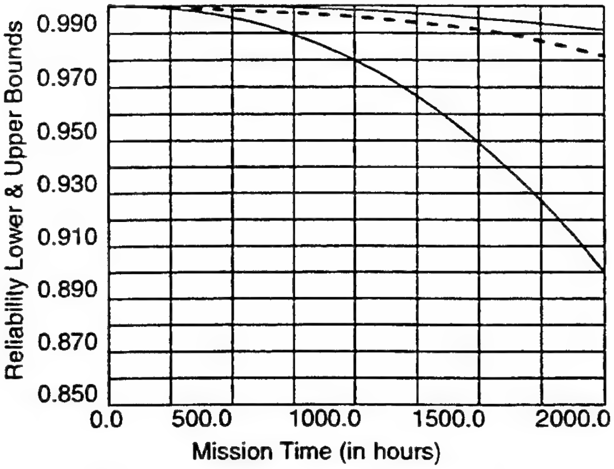
### 4.5.2 FTPP System

This section presents results for the Fault-Tolerant Parallel Processor (FTPP). The Fault-Tolerant Parallel Processor (FTPP) is a byzantine-resilient computer architecture which uses a number of processing elements operating in redundant groups to achieve both high reliability and high throughput [174]. A sample 4-NE, 16-PE FTPP cluster is shown in Figure 30. In this example, there are 3 Triad groups (T1-T3), 1 Quad group (Q1), and 3 Simplex computers (which could be used as spares). Triad 1 (T1) is formed from PEs on NEs 1, 2, and 3. Triad 2 (T2) is formed from PEs on NEs 1, 2, and 4. Triad 3 is formed from PEs on NEs 2, 3, and 4. Quad 1 (Q1) is formed from a PE from each NE. Finally, NEs 1, 3, and 4 contain a simplex PE which

Appeared in the Proceedings of the AIAA Computing in Aerospace Conference, 1995, pp. 285-300



(a) Reconfigurable Quad with $\lambda$= 0.00005



(b) Reconfigurable Quad with $\lambda$ = 0.0001

**Figure 29. Quad REST/Simulation Results**

could be used as a spare for failed PEs within the various FMGs. The results obtained using the ADEPT-REST interface are shown in Figure 31.

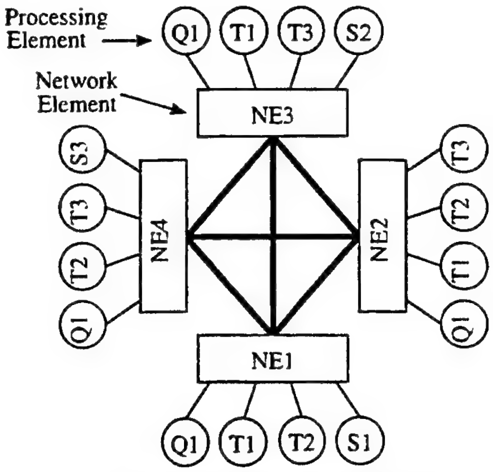


**Figure 30. Sample FTPP Cluster**

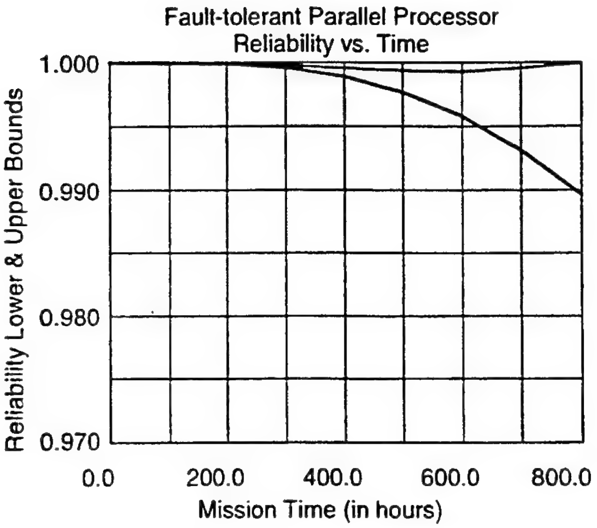


**Figure 31. FTPP ADEPT-REST results**

## 5. Conclusions

This paper has described a unified design environment called ADEPT which supports the design of complex systems from initial concept to final implementation. Both simulation-based and analytic approaches can be utilized for performance and reliability analysis. In addition to providing an integrated environment for high level performance and reliability analysis, ADEPT also supports the stepwise refinement models.

This paper has also demonstrated the benefit of having an underlying mathematical foundation for the ADEPT modules, that of colored Petri Nets. Using Petri Net reduction techniques, the system model can be reduced in order to speed up simulation. Further, the colored Petri net descriptions can also be used to construct Markov models from which reliability and safety information can be derived. A methodology to interface the REST engine with the ADEPT-VHDL model was also presented with examples and results.

Current research includes incorporating importance sampling techniques for the simulation based approach to reliability analysis. We are also investigating techniques for incorporating safety, availability, and performability analysis into ADEPT. Automated generation of fault trees from ADEPT models and solution techniques using Binary Decision Diagrams (BDDs) are also being investigated.

## 6. References

[139]J. H. Aylor. R. Waxman, B. W. Johnson, R. D. Williams, "The Integration of Performance and Functional Modeling in VHDL," in *Performance and Fault Modeling with VHDL*, J. M. Schoen (Ed.), Prentice-Hall, Englewood Cliffs, NJ, 1992.

[140]K. Jensen. "Colored Petri Nets: A high level language for system design and analysis," in *High-level Petri Nets:*

*Theory and application,* K. Jensen and G. Rozenberg (Eds.), Berlin: Springer-Verlag, 1991, pp. 44-119.

[141]F. T. Hady, *A Methodology for the Uninterpreted Modeling of Digital Systems in VHDL,* Master's Thesis, Dept. of Electrical Engineering, University of Virginia, January 1989.

[142]G. Swaminathan, R. Rao, J. H. Aylor, and B. W. Johnson, *Colored Petri Net descriptions for the UVA primitive modules,* CSIS Technical Report 920922.0, University of Virginia, September 1992.

[143]J. B. Dennis, "Modular, Asynchronous Control Structure for a High Performance Processor," *ACM Conference Record, Project MAC,* Massachusetts, 1970, pp. 55-80.

[144]E. D. Cutright, R. Rao, B. W. Johnson, J. H. Aylor, *A Handbook on the Unified Modeling Methodology Building Block Set*, CSIS Technical report 910828.2, University of Virginia, August 1993.

[145]Mentor Graphics Corporation, *Design Architect Reference Manual,* Wilsonville, OR, Vol. I and II, Version 8.1, 1992.

[146]IEEE, *"IEEE Standard VHDL Language Reference Manual,"* New York, NY, IEEE Std. 1076-1987, March 31, 1988.

[147]W. A. Wulf, "Experimental Implementation of Dynamic Access Ordering," *Proceedings of the 27th Annual Hawaii International Conference on System Sciences,* 1994, pp. 431-440.

[148]Vantage Analysis Systems, *VantageSpreadsheet User's Guide,* Oakland, CA, Vol. 1, 1992.

[149]R. Rao, B. W. Johnson, J. H. Aylor, G. Swaminathan, "Synthesis of Reliability Models from Behavioral Performance Models," *1994 Reliability and Maintainability Symposium (RAMS),* Anaheim, CA, January 1994.

[150]E. D. Cutright and B. W. Johnson, "A Simulation-based Approach to Integrated Performance and Reliability Modeling using VHDL," *1994 Reliability and Maintainability Symposium (RAMS),* Anaheim, CA, January 1994.

[151]D. Nicol, et al., *Users Guide to the Reliability Estimation System Testbed (REST),* NASA Technical Memorandum 107596, June 1992.

[152]G. Swaminathan, R. Rao, J. H. Aylor, B. W. Johnson, "A VHDL Based Environment for System Level Design and Analysis," *Proceedings Spring 1994 VHDL International User's Form,* Oakland, CA, May 1-4, 1994, pp. 110-116.

[153]G. Swaminathan, J. H. Aylor, and B. W. Johnson, *An* $O(n^2 \log n)$ *Time Petri Net Reduction Algorithm,* CSIS Technical report 910805.0, University of Virginia, August 1991.

[154]R. D. Williams, *Spring 1988 EE735 Class Project,* University of Virginia, 1988.

[155]R. Rao, *A Building Block Approach to Performance Modeling using VHDL,* Master's Thesis, Dept. of Electrical Engineering, University of Virginia, May 1990.

[156]P. J. Hayes and A. Andrews, "Multiprocessor Performance Modeling with ADAS," *Proceedings of the Seventh AIAA Computers in Aerospace Conference,* Monterey, CA, October 1989.

[157]E. D. Cutright, R. Rao, B. W. Johnson, and J. H. Aylor, *Modeling an ATAMM Based Multiprocessor System using VHDL,* CSIS Technical Report 910111.0, Dept. of Electrical Engineering, University of Virginia, January 1991.

[158]P. F. Reynolds Jr., C. M. Pancerella, and S. Srinivasan, "Design and Performance Analysis of Hardware Support for Parallel Simulations," *Journal of Parallel and Distributed Computing,* Vol. 18, August 1993, pp. 435-453.

[159]S. Mitra, *An Interpretation Interface for Hybrid Modeling,* Master's Thesis, Dept. of Electrical Engineering, University of Virginia, January 1992.

[160]A. Sarkar, R. Waxman, J. P. Cohoon, "System Design Utilizing Integrated Specification and Performance Models," *Proceedings Spring 1994 VHDL International User's Form,* Oakland, CA, May 1-4, 1994, pp. 90-99.

[161]D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming,* Vol. 8, No. 3, 1987, pp. 231-274.

[162]S. Kumar, J. H. Aylor, B. W. Johnson, W. A. Wulf, "A Framework for Hardware/Software Codesign," *IEEE Computer,* Vol. 26, No. 12, December 1993, pp. 39-45.

[163]S. Kumar, R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, R. Waxman, "ADEPT: A Unified System Level Modeling Design Environment," Proceedings of the 1st Annual RASSP Conference, Arlington, Virginia, August 15-18, Virginia, pp. 114-123.

[164]A.Z. Keller, "Monte Carlo Simulation in Reliability," *Reliability Modeling and Applications*, ECSC, 1987, pp. 59-69.

[165]Norman McCormick, *Reliability and Risk Analysis*, Academic Press, Inc., 1981.

[166]B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, Reading, Massachusetts, 1989.

[167]R. Geist and K. Trivedi, "Reliability Estimation of Fault-Tolerant Systems," *IEEE Computer*, July 1990, pp. 52-61.

[168]A. Goyal, et. al., "A Unified Framework for Simulating Markovian Models of Highly Dependable Systems", *IEEE Trans. on Computers*, Vol. 41, Jan. 1992, pp. 36-51.

[169]R.W. Butler, "The SURE Approach to Reliability Analysis", *IEEE Transactions on Reliability*, Vol. 41, No. 2, June 1992, pp. 210-218.

[170]J. B. Dugan, et. al., *Extended stochastic Petri nets: Applications and analysis*, Performance '84 Models of Computer System Performance. North Holland, Amsterdam, 1985.

[171]B. Beyaert, et. al., *Evaluation of computer system dependability using stochastic Petri nets*, Digest of the 11th Annual Symposium on Fault-Tolerant Computing (IEEE), July 1981, pp. 79-81.

[172]H. H. Ammar, *Hierarchical Models for Systems Reliability, Maintainability, and Availability*, IEEE Transactions on Circuits and Systems, Vol. 34, No. 6, June 1987, pp. 629-638.

[173]M. K. Molloy, *Performance Analysis Using Stochastic Petri Nets*, IEEE Transactions on Computers, Vol. 31, No. 9, Sept. 1982, pp. 913-917.

[174]R. Harper, J. Lala, "Fault-Tolerant Parallel Processor," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 14, No. 3, May/June 1991, pp. 554-563.

**Appeared in the Proceedings of the AIAA Computing in Aerospace Conference, 1995, pp. 285-300**

# Dependable System Codesign Using Data Flow Models

Charles Y. Choi • University of Virginia • Charlottesville
Barry W. Johnson • University of Virginia • Charlottesville
Joanne Bechta Dugan • University of Virginia • Charlottesville

**Key Words:** Hardware/Software Codesign, Data Flow, System Design, Dependability, Rapid Prototyping

## *SUMMARY & CONCLUSIONS*

The need to account for both spatial and temporal redundancy in dependable system design requires the use of abstractions that model both hardware and software. To answer this, we present a novel approach to designing dependable systems by using hardware/software codesign request-resource models. Out of this work we aim to demonstrate how dependability analysis can be embedded into the design cycle, a currently difficult situation due to the difference in paradigms used to construct systems as opposed to analyzing them.

A framework for dependable system design is presented to show how codesign models can be generated using rapid prototyping techniques. This framework is implemented in a design environment called ADEPT (ADvanced Environment Prototype Tool). Codesign models are built from a library of nodes that adhere to a data flow model of computation. The prototype codesign models can then be analyzed for their functional, performance, and dependable characteristics. An example system using a 3N code was modeled to demonstrate the utility of the framework in doing trade-off analysis during its design.

## *1. INTRODUCTION*

The need to account for spatial and temporal redundancy in dependable system design is being exacerbated by the increase in scale and usage of such systems. With regards to the three universe model shown in Figure 1 [1], spatial redundancy exists in the physical universe, whereas temporal redundancy exists in the informational. However, current modeling approaches used to obtain dependability metrics are not able to reconcile both forms of redundancy, chiefly due to their basis in evaluation oriented state based models [2]. State based models such as Markov chains and fault trees poorly model the interaction between the physical and informational universes, at best implicitly acknowledging the two. Furthermore, most design methodologies dismiss dependability analysis until late in the design cycle because the paradigms used to model system construction are different from those used to model system dependability. System construction uses information flow based models, where a set of tasks are interconnected to send information from one task to another. System dependability analysis uses state based models to obtain metrics such as reliability, safety, and availability. The different paradigms have made it difficult to appropriately design systems for life-critical applications where dependable features must be embedded in the architecture of a system. In short, a different approach to modeling dependable systems is needed.

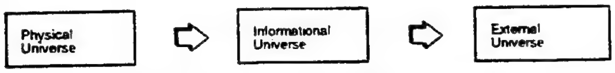


*Figure 1* Three universe model

Bridging the physical with the informational universe is a central issue in hardware/software codesign (or codesign). This paper asserts that work in codesign can help in modeling both spatial and temporal redundancy in a unified fashion. We also feel that within codesign lies the key to embedding dependability analysis into the design cycle. To demonstrate this assertion, this paper will explore ways of modeling dependable systems using codesign models [3]. To construct the models we use data flow nodes. Of particular interest in using a data flow model of computation is its functional properties which allow for equational/formal reasoning [4]. Furthermore, model construction using data flow nodes is simple and easily extensible, making it an ideal choice for rapidly prototyping systems and comparing candidate architectures.

## *2. THE TROUBLE WITH REQUIREMENTS CAPTURE*

System design is largely an exercise in determining and meeting requirements. However, capturing requirements has been a persistently hard problem due to the lack of complete information in either the context or behavior of system operation. In addition, requirements has been historically viewed as a "what versus how" problem: requirements state *what* the system will do without referring to *how* it will do it. The flaw with this view is its simplicity: requirements and design are in practice interdependent; the "what versus how" mentality is a reflection of a management decision to partition those who conceive of an idea from those who implement it [5]. Current work in requirements has yielded an alternative view that is described in terms of domain - requirements - machine. The domain provides context for requirements which are satisfied by the machine. The argument for this view is that information is situated and situations determine the meaning of requirements.

**Appeared in the Proceedings of the Reliability and Maintainability Symposium, 1997**

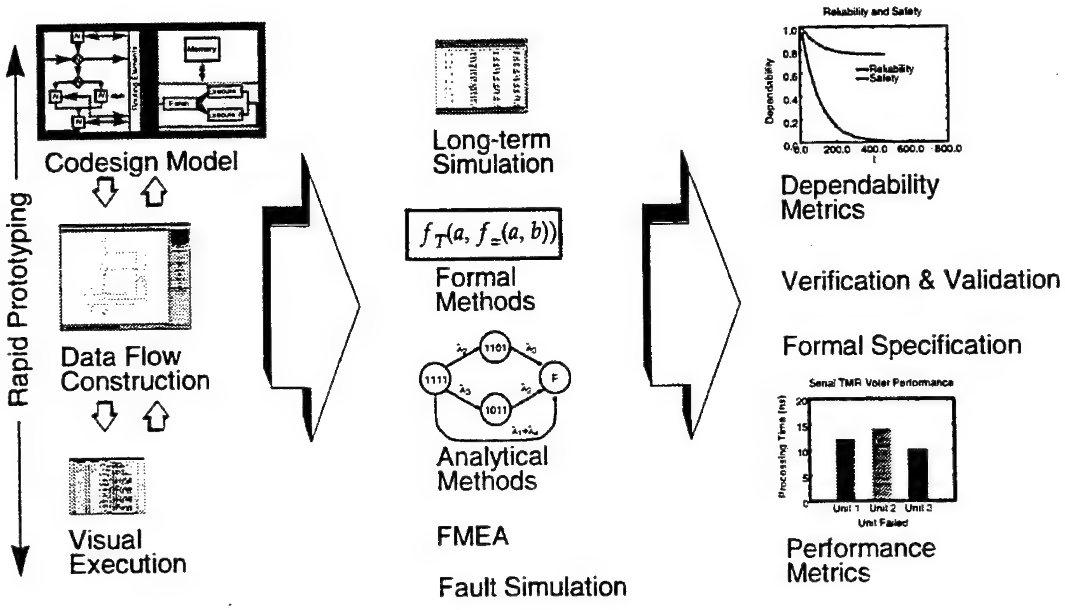


*Figure 2* A framework for dependable system codesign

In general, dependability analysis has taken a reductionist view of systems, measuring attributes such as reliability and availability as the aggregate quantity of a system's components. In [6], Leveson asserts that this approach to constructing safe systems is less than desirable, arguing that safety derives from an emergent property of component interactions, not from the properties of components. Safety critical design in this light entails study of systemic hazards that can either be designed out or minimized as much as humanly possible. Put another way, safety critical design lies in the realm of defining and analyzing the requirements of systems.

## 3. A FRAMEWORK FOR DEPENDABLE SYSTEM DESIGN

A technically feasible response to the persistently hard problem of comprehensive requirements capture is to avoid it altogether: instead use rapid prototyping to flesh out assumptions and verify system behavior [7]. In [8] Gordon and Bieman go over 39 case studies that illustrate the benefits of rapid prototyping to system design. This paper suggests that rapid prototyping is necessary to develop dependable system requirements to compensate for incomplete information on context and that data flow nodes are a natural means for tracing these requirements. Figure 2 shows a framework for dependable system codesign. In this framework, systems are represented using a codesign request-resource model coupled with software architectures taxonified by [9]. Dependable strategies can then be implemented within the framework of this request-resource model to satisfy the notion of dependable system codesign. As the codesign model is constructed and refined using data flow nodes, virtual prototypes are generated. These prototypes can be executed and visually inspected to ensure their correctness. Prototypes can be analyzed using a number of simulation and analytical techniques. Fault simulation can be used to verify dependable strategies and measure dependability. Simulation can also provide performance information such as throughput and latency. The data flow graph could be mapped to a Markov chain for solution or be mapped to term equations such that formal techniques can be used to verify and validate behavior as well as formally specify its function. As the prototypes are refined over different levels of abstraction, more accurate measures of performance and dependability metrics can be obtained.

Research at the University of Virginia has developed an design environment called ADEPT. The idea underlying ADEPT is to provide a means to do performance and dependability analysis on a single design model that can be refined over multiple levels of abstraction. In practice, one develops a model by visually constructing a graph using a schematic capture tool. This resulting graph is then translated into VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language), whereupon the model can be simulated. ADEPT relies on VHDL's capability to define and model designs at multiple levels of abstraction. The implementation of our framework for dependable system codesign consists of a library of data flow nodes and a set of post-processing tools. Note that example codesign models shown later in the paper are the actual schematics of models constructed within ADEPT.

### 3.1 *Data Flow*

It is common to describe systems at a high level as a task graph, where information is transferred from one task to another, each task performing some transformation on that information. Data flow modeling is an ideal candidate for representing such systems. The data flow modeling concept stems primarily from work by Dennis [10] as a means for organizing large scale non-sequential computations. A considerable amount of literature on data flow modeling exists; [4], [11], and [12] provide good tutorial presentations on data flow theory and mechanics. Data flow is a functional model of computation characterized by: 1) a control discipline based on the availability of operand objects; 2) an operational discipline based on the orderly consumption and (re-)production of operand objects; 3) the realization of primitive and defined functions as constant operator objects [11]. The idea behind functional computation (and thus data flow computation) is to map the computation as closely to mathematical functions and functional composition as possible. As such, purely functional programs do not use temporary variables nor assignment statements, rather functional programs use function definitions and function application specifications. The execution of a program is accomplished by evaluating the function. A common representation for a data flow model is a directed graph which depicts operators as nodes and operands as tokens that traverse the arcs interconnecting them.

The firing rule generally follows the same behavior of Petri nets: once all the inputs to a node are filled with tokens, the node consumes all of the input tokens and places token(s) on its output(s). Pure data flow graphs also share the same attribute with simple Petri nets in that they are not Turing complete. Extensions based on the inhibitor arc provide data flow graphs Turing completeness; however such extensions diminish the decideability of data flow graphs that use them.

### 3. 2 *Hardware/Software Codesign*

Hardware-software codesign (or codesign) has been defined as the integrated design of systems implemented using both hardware and software components [13]. Recent interest in this field has arisen from advances in methodologies that concurrently apply and trade-off design techniques developed from both domains [3]. Further driving work in codesign has been the desire to better manage the current design of embedded systems through the use of higher levels of design abstraction [17][18]. Initially, codesign is a study in the methodology of system development, driven by the deficiencies of current approaches. A common practice is to partition the hardware and software development paths early in the design cycle, often developing the hardware first and then writing the software to run on it as shown in Figure 3. Later, during the system integration phase, both the hardware and the software is tested as a whole. Problematic with this methodology is the early divergence of hardware and software development. Hardware can be developed without consideration for software requirements such as processing speed and memory size. Software can be developed without knowledge of changes made to the hardware during its development. As a result, integration late in the design cycle can incur significant cost increases and schedule overruns.

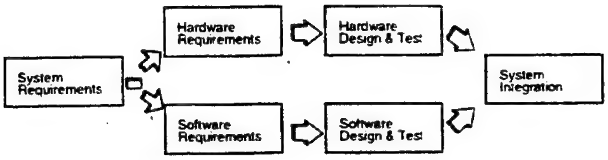


*Figure 3* Current system development methodology

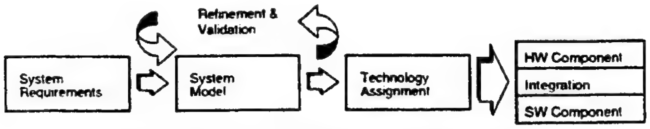


*Figure 4* Late binding methodology

An alternative approach is to model the system requirements as fully as possible and defer the hardware/software partitioning as late as possible in the design cycle. This is referred to in the codesign community as late binding. Such a methodology, as shown in Figure 4, relies on a unified design representation of hardware and software. Attributes of such a representation include the ability to evaluate both hardware and software in a common simulation environment and the ability to easily transfer functionality between the two domains. In [14], Kumar presents a unified representation of hardware and software using a graph model based on requests and resources. In a request/resource model, a program is a sequence of requests. These requests are serviced by a finite set of resources. As such, software and hardware are mapped to requests and resources respectively where operations in software utilize resources within the hardware and manipulate its state. The model structure used by Kumar entailed decomposing the system into a software and hardware model as shown in Figure 5.

In this model, software is modeled using a combination of process and predicate nodes. Process nodes repre-

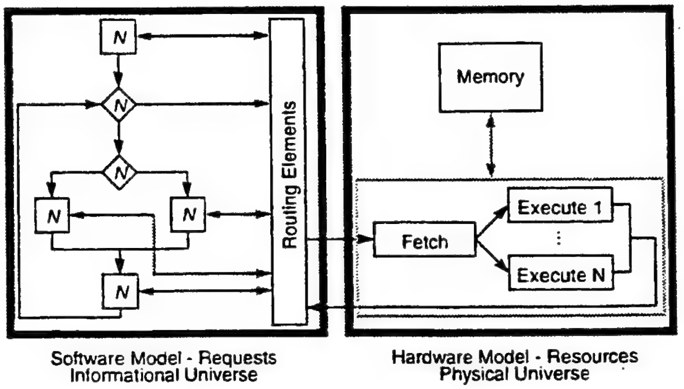


*Figure 5* Modeling structure for an abstract codesign model [14]

sent functional transformations such as arithmetic operations. Predicate nodes represent conditional or decision structures such as if-then-else clauses or while loops. Both the process and predicate nodes represent computations that can make requests upon the hardware model.

Hardware is modeled using a combination of resource nodes and predicate nodes. The resource nodes represent functional units such as ALU's, processors, or memory elements. A common approach to modeling hardware is to emulate the fetch-execute cycle. So as both the hardware and the software models are joined together, the requests generated by the software model can be serviced by the hardware model using a fetch-execute cycle.

### 3. 3 *Modeling Dependable Strategies*

Two approaches to increasing the dependability of a system are to use fault avoidance and fault tolerance. Fault avoidance lies in the realm of requirements capture: for this we rely on rapid prototyping to check for and design out potential hazards. Fault tolerance relies on redundancy to add information to offset the effect of faults. Going back to our categorization of redundancy as either spatial or temporal, spatial redundancy uses the addition of extra components to accommodate added information. Temporal redundancy uses extra time to recompute or recheck computations. In the context of our codesign model, spatial redundancy can be accomplished by concurrently replicating resources. Temporal redundancy can be accomplished by concurrently replicating requests and mapping them to a single resource.

There is a wealth of information on design strategies to improve dependability. Three texts which go over this topic in detail are [15], [16], and [1]. Rather than reiterate their writings on the implementation of dependable

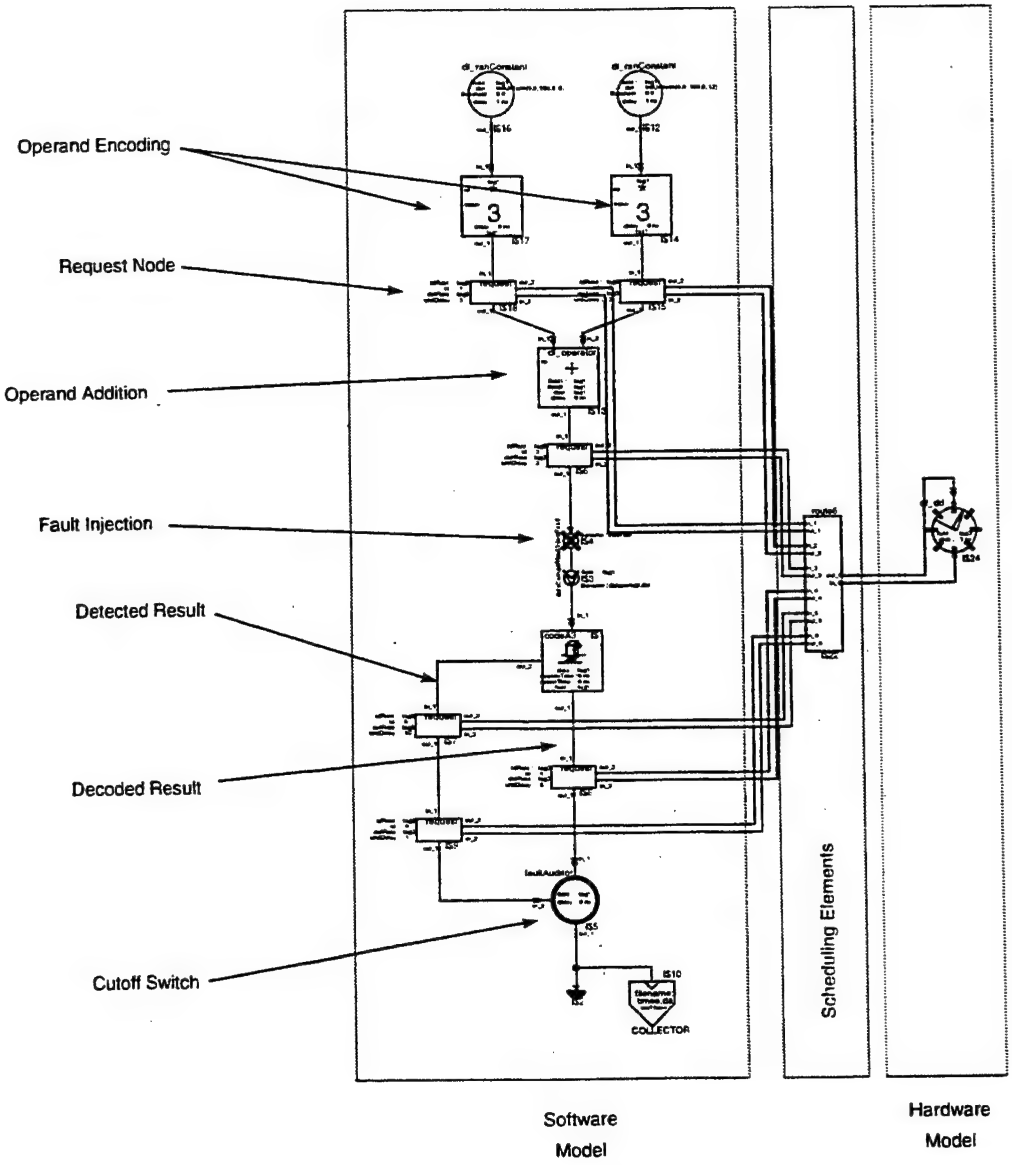


*Figure 6* Codesign model of a simplex architecture with 3N coding

**Appeared in the Proceedings of the Reliability and Maintainability Symposium, 1997**

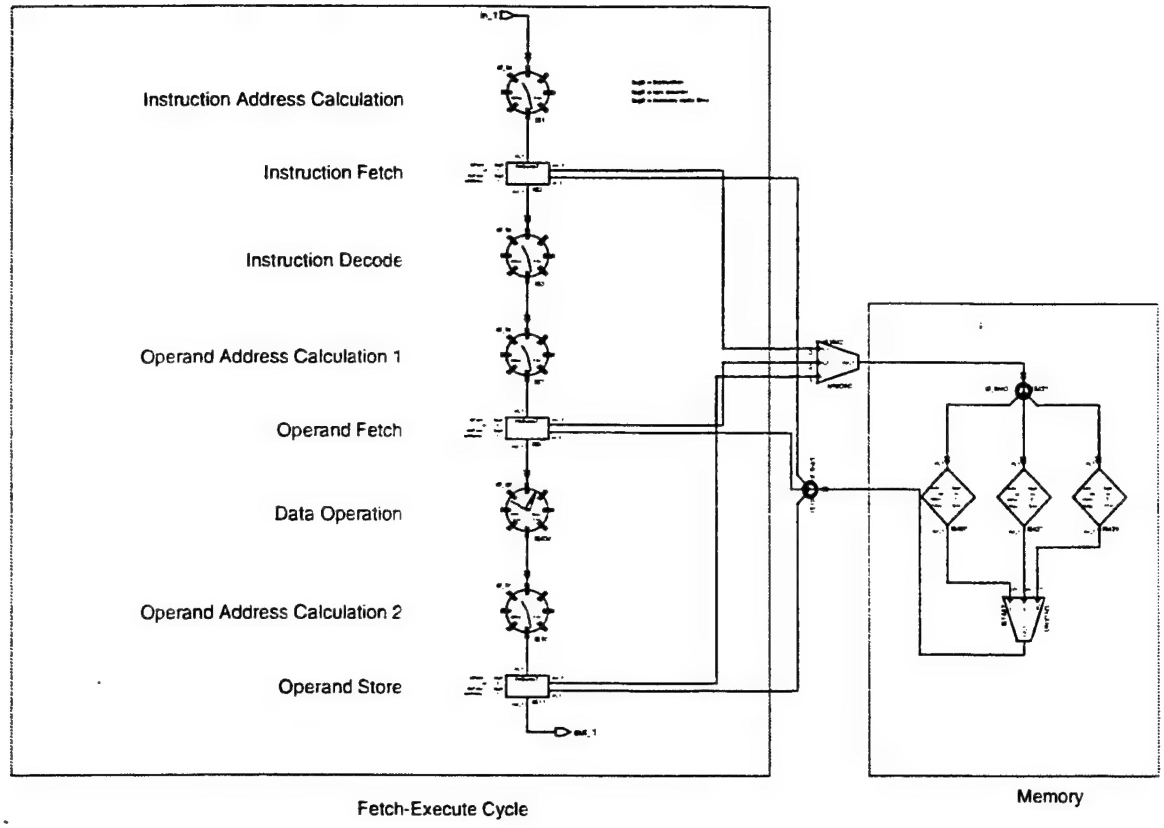


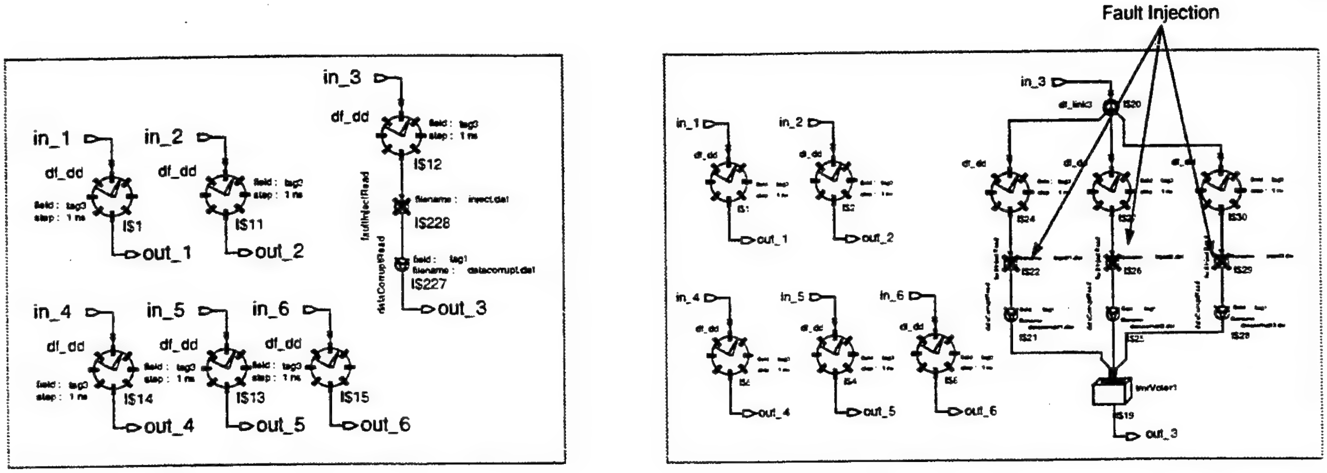
*Figure 7* Example hardware model representing a fetch-execute cycle

strategies, we will go over in brief the objectives that all dependable systems embody and examples to illustrate a dependable codesign approach. Redundant systems may go through as many as ten of the following steps to respond to the occurrence of a fault [15]: 1) *Fault containment*, where techniques used to localize the effect of a fault are used to protect the rest of the system; 2) *Error detection*, where efforts to determine if a fault has occurred are exercised; 3) *Fault masking*, where the occurrence of faults are hidden in such a manner as to prevent an error from occurring; 4) *Retry*, where successive attempts at computation are done to obtain a successful result. This is particularly useful in overcoming transient faults that cause no physical damage; 5) *Diagnosis*, where an a priori sequence of steps are taken to determine the correct operation of a system; 6) *Reconfiguration*, where a component with a permanent fault which has been detected and located can be replaced and/or isolated from the rest of the system; 7) *Recovery*, where after detection and if necessary, reconfiguration, the effect of the error is eliminated. This is often accomplished using rollback, where system operation is backed up to a point preceding the error detection and recommences operation; 8) *Restart*, where internal state information may be so corrupted that resetting the system is the only recourse of action; 9) *Repair*, where a component that is diagnosed as failed is replaced. This can either be done in an on-line or off-line fashion; 10) *Reintegration*, where after a component is physically replaced it is brought back in the function of the system.

[1] describes in detail four approaches towards implementing redundancy, these being *hardware*, *software*, *information*, and *time*. Three basic forms of hardware redundancy are passive, active, and hybrid. Passive approaches use fault masking which is typically accomplished using voting. Active methods use error detection, location, and recovery to maintain resiliency. Hybrid approaches combine both active and passive techniques. Software redundancy uses extra software to detect and possibly tolerate faults. Consistency checking, capability checking, and $N$-version programming are all examples of software redundancy. Information redundancy uses added information beyond what is necessary to implement a certain function. Examples of this include error detecting and correcting codes. Time redundancy uses extra time to perform a function such that fault detection and often fault tolerance can be achieved.

## *4. AN EXAMPLE 3N CODE SYSTEM*

To demonstrate the utility of this framework, we apply it to an example 3N code system shown in Figure 6. This diagram is partitioned into three sections: on the left is the software model, on the right is the hardware model and in between them are scheduling elements. For each task in the software model there is a request node



Operand Addition Simplex Architecture

Operand Addition TMR Architecture

*Figure 8* Comparison of simplex and TMR architectures within candidate hardware prototypes

which calls on a resource in the hardware model. Since we consider software to be a timeless informational quantity, no delay is assigned to the nodes in the software model. Delay in the overall model is accounted for in the hardware model, where delay is a reflection of the physical world. Each of the nodes contained within both models is comprised of either a data flow node or a hierarchical collection of data flow nodes. At its simplest, the resource model can be represented as a data dependent delay that reflects the cost associated with each request. If further detail is desired, the hardware model can be refined into a fetch-execute model shown in Figure 7. Of note is how the fetch-execute model reflects on the higher level request-resource model; the fetch and execute processes are mapped to requests and the memory serves as the resource.

## *5. SYSTEM ANALYSIS*

With the codesign model, performance trade-offs can be made by scheduling concurrent software tasks to multiple hardware elements. In our 3N code example, six requests are made by the software model to perform operand encoding (2 requests), operand addition (1 request), error detection (1 request), decoding (1 request), and switching (1 request). Of these requests, operand encoding and error detection and decoding can be mapped onto separate hardware components for concurrent execution. To analyze our system we will construct three different prototypes of our 3N code model. In one prototype, all six requests will be mapped onto a single resource as shown in Figure 6. In the two other prototypes, each request has its own resource, thus exploiting all of the concurrency available in the software model. However, what distinguishes the latter two is the architecture used to serve the operand addition request (denoted by the input-output pair (in_3, out_3). One prototype shown in Figure 8 uses a simplex architecture. The other, also shown in the same figure, uses a triple modular redundant (TMR) architecture. Since both the informational and physical universes are represented in the codesign model, both software and hardware faults can be represented on the resource and the request sides respectively.

Simulation of the codesign models in VHDL provided performance and dependability information. However, a caveat: no effort was made to generate statistically valid results much less provide a detailed comparative analysis of the example architectures; rather our intent is to illustrate how one can use codesign to rapidly make trade-off decisions with both performance and dependability metrics. Figure 9 shows the performance of the three prototypes in terms of latency and throughput. From the performance result, we see that the single resource hardware model had the longest latency, which is expected since no concurrency was exploited. The TMR model had longer latency than the simplex model due to the delay required to vote on the three additions. Also due to concurrency was the higher throughput of the simplex and TMR prototypes compared to the single resource prototype.

Fault simulation was used to measure the reliability and safety of the prototypes. Using permanent faults, data was corrupted such that the operand values had added to them values of either 0, 1, or 2. The 3N coding scheme would be able to detect faults that were off by 1 or 2. Injection times were exponentially distributed, with a failure rate $\lambda = 0.01$ . Faults that did not change the value of an operand were tracked so that unsafe failures could be accounted for. For the single resource prototype, we decided to inject the fault in the software after the operand addition. For the simplex and TMR prototypes, we injected fault(s) in the resource(s) that serviced the operand addition. Simulating each prototype 1000 times, failure times were tallied so that plots of reliability and safety could be generated as shown in Figure 10. From the results we see that the simplex and TMR prototypes do exhibit the expected crossover behavior in their reliability plots. Furthermore, we see that the software fault in the single resource prototype gives that system a lower level of reliability than either of the other two. The simplex exhibited the highest level of safety of the three models, followed by the TMR and then the single resource. However, later on the safety of the TMR prototype drops below that of the single resource.

## 6. CONCLUSIONS

This work started with the desire to develop abstract models to account for spatial and temporal redundancy in a unified fashion. To this end a framework for dependable system design has been created, where systems are rapidly prototyped using codesign request-resource models. This framework is implemented in a design environment called ADEPT. Codesign models are built from a library of nodes that adhere to a data flow model of computation. The prototypes can then be analyzed for their functional, performance, and dependable characteristics. An example system using a 3N code was modeled to demonstrate the utility of the framework in doing trade-off analysis during its design.

## ACKNOWLEDGEMENTS

The authors would like to acknowledge the support provided by the Defense Advanced Research Projects Agency under contract number F33615-93-C-1313.

## REFERENCES

1. B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, pp. 584, 1989.
2. J.F. Meyer, "Performability: a retrospective and some pointers to the future", *Performance Evaluation 14*, Elsevier Science s, 1992, pp. 139-156.
3. K. Buchenrieder, J. W. Rozenblit, *Codesign: Computer-Aided Software/Hardware Engineering*, IEEE Press, 1995.

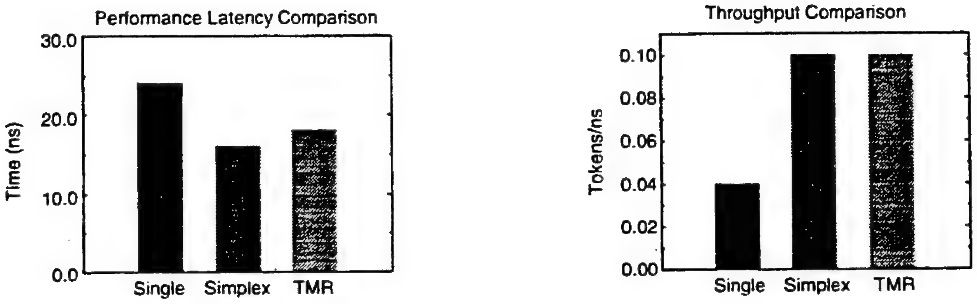


*Figure 9* Performance results of single resource, simplex, and TMR prototypes

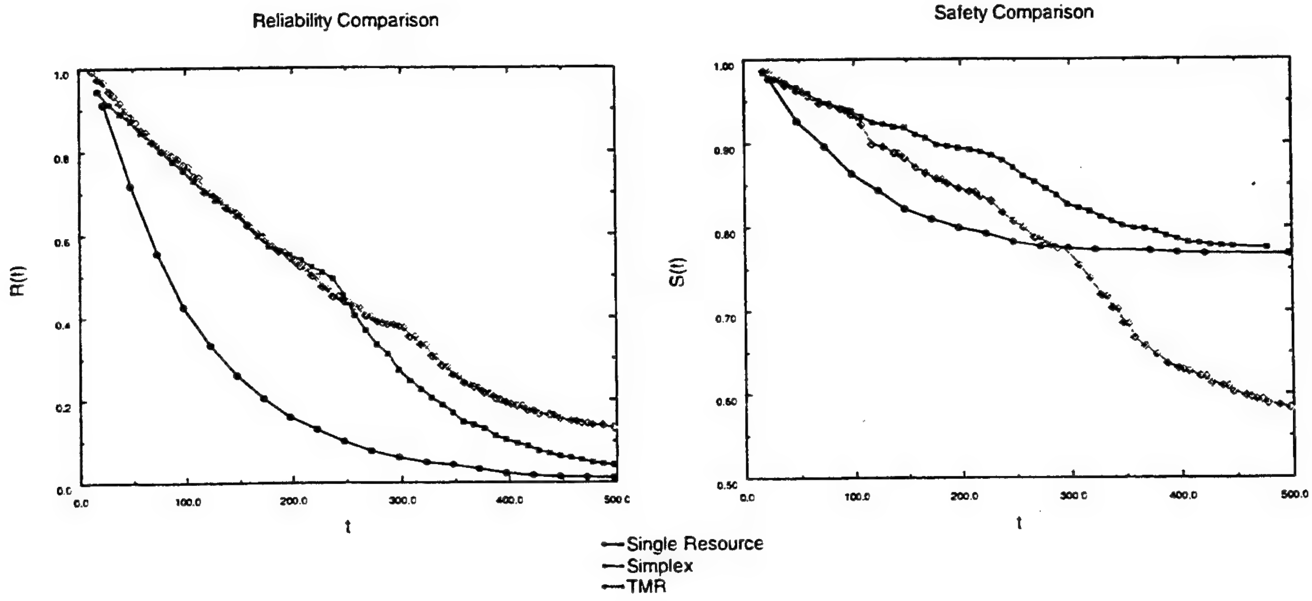


*Figure 10* Dependability results of single resource, simplex, and TMR prototypes

**Appeared in the Proceedings of the Reliability and Maintainability Symposium, 1997**

4. J. A. Sharp, *Data Flow Computing*, Ellis Horwood Limited, 1985.
5. J. Siddiqi and M.C. Shekaran, "Requirements Engineering: The Emerging Wisdom", *IEEE Software*, March 1996, pp. 15-19.
6. N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley Publishing Company, Inc., 1995.
7. T. Lewis, "The Next $10000_2$ Years: Part II", *Computer*, May 1996, v. 29, no. 5, pp. 78-86.
8. V.S. Gordon and J.M. Bieman, "Rapid Prototyping: Lessons Learned", *IEEE Software*, Jan. 1995, pp. 85-95.
9. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
10. J.B. Dennis, "First Version of a Data Flow Procedure Language", *Proc Colloque sur la Programmation*, Springer Verlag, 1974, ed. B. Robinet, v. 19, Lectures in Computer Science, Heidelberg, pp. 362-376.
11. W. Kluge, *The Organization of Reduction, Data Flow and Control Flow Systems*, The MIT Press, 1992.
12. S. Dasgupta, *Computer Architecture: A Modern Synthesis*, John Wiley & Sons, 1989.
13. P. Subrahmanayam, "Hardware-Software Codesign: Cautious Optimism for the Future (Hot Topics)", *Computer*, January 1993, v. 26, no. 1, pp. 84-85.
14. S. Kumar, *A Unified Representation for Hardware/Software Codesign*, Ph.D. Thesis, University of Virginia, 1995.
15. D.P. Siewiorek and Robert S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, 1982.
16. D.K. Pradhan, *Fault Tolerant Computer System Design*, Prentice-Hall, 1996
17. R.K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, 1995.
18. D.D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice-Hall, 1994.

## *BIOGRAPHIES*

**Charles Y. Choi**
Department of Electrical Engineering
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903-2442 USA
*Internet*: choi@virginia.edu
*Voice*: 804.982.2965 (w) / 804.296.6186 (h)
*Fax*: 804.924.8818

Charles Yong Choi received his B.A. in Physics and M.S. in Electrical Engineering at the University of Virginia in 1987 and 1992, respectively. At current he is a Ph.D. candidate in Electrical Engineering at the University of Virginia. His research interests include ESDA methodologies and tools, fault tolerance, data flow modeling, and computer graphics. Mr. Choi is a member of the IEEE Computer Society, Eta Kappa Nu, and the Raven Society.

**Barry W. Johnson**
Department of Electrical Engineering
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903-2442 USA
*Internet*: bwj@virginia.edu
*Voice*: 804.924.7623 (w) / 804.978.1096 (h)
*Fax*: 804.924.8818

Barry W. Johnson is currently a Professor in the Department of Electrical Engineering at the University of Virginia. He is also a cofounder and the Director of the Center for Semicustom Integrated Systems, a Technology Development Center of the Virginia Center for Innovative Technology. Prior to joining the University, he was with Harris Corporation in Melbourne, Florida where he participated in the design and analysis of fault-tolerant computer systems for aerospace applications. His research and teaching interests include fault-tolerant computing, safety-critical systems, VLSI architectures, VLSI testing, and microprocessor-based systems. He is the author of a textbook entitled The Design and Analysis of Fault-Tolerant Digital Systems which is published by Addison-Wesley Publishing Company. He is also the author of more than 75 journal and conference articles. Johnson is presently active in the IEEE Computer Society as the President-elect, a member of the Executive Committee, and a member of the Board of Governors. Previously, he has served as the IEEE Computer Society's Vice President for Publications, Vice President for Conferences and Tutorials, Vice President for Press Activities, and Vice President for Membership.

Johnson received the BS, ME, and Ph.D. degrees in electrical engineering from the University of Virginia, Charlottesville, Virginia, in 1979, 1980, and 1983, respectively. He is a Fellow of the IEEE and a member of the IEEE Computer Society, Tau Beta Pi, Eta Kappa Nu, and Sigma Xi. He has received several awards including the 1992 C. Holmes MacDonald Outstanding Young Electrical Engineering Professor Award from Eta Kappa Nu, the 1991 Frederick Emmons Terman Award from the American Society for Engineering Education, a 1992 Alan Berman Research Publications Award from the Department of the Navy, and a 1990 Outstanding Faculty Award from the State Council of Higher Education in Virginia.

**Joanne Bechta Dugan**
Department of Electrical Engineering
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903-2442 USA
*Internet*: jbd@virginia.edu
*Voice*:804.982.2078 (w) / 804.823.7865 (h)
*Fax*: 804.924.8818

Joanne Bechta Dugan was awarded the B.A. degree in Mathematics and Computer Science from La Salle University, Philadelphia, PA in 1980, and the M.S. and Ph.D. degrees in Electrical Engineering from Duke University, Durham, NC in 1982 and 1984, respectively. Dr. Dugan is currently Associate Professor of Electrical Engineering at the University of Virginia, and was previously Associate Professor of Computer Science at Duke University and Visiting Scientist at the Research Triangle Institute. She has performed and directed research on the development and application of techniques for the analysis of computer systems which are designed to tolerate hardware and software faults. Her research interests thus include hardware and software reliability engineering, fault tolerant computing, and mathematical modeling using dynamic fault trees, Markov models, Petri nets and simulation. Dr. Dugan is an Associate Editor of the IEEE Transactions on Reliability, is a Senior member of the IEEE, and is a member of Eta Kappa Nu and Phi Beta Kappa. She is serving on the National Research Council Committee on Application of Digital Instrumentation and Control Systems to Nuclear Power Plant Operations and Safety.

**Appeared in the Proceedings of the Reliability and Maintainability Symposium, 1997**

# Hybrid Modeling with Synchronous Interpreted Elements

Moshe Meyassed
Robert H. Klenke
James H. Aylor
Department of Electrical Engineering
University of Virginia

**Center For Semicustom Integrated Systems
Department of Electrical Engineering
University of Virginia
Charlottesville, Virginia 22903-2442**

**CSIS Technical Report #980317.0
March 17, 1998**

# 1 Introduction

The rapid growth in the complexity of digital systems is creating the need for better and more efficient design tools and methods. As the complexity of a system increases, so will the need for design automation at more abstract levels where trade-offs are easier and quicker to perform and understand [1][2]. Therefore, as a part of the pre-synthesis phase of the design cycle, a high level performance model is usually constructed and simulated. This modeling level provides the ability to check different possible architectures and to estimate whether a particular architecture will meet the system requirements. Following this stage, analysis at more detailed design level such as Register Transfer Level (RTL) or behavioral level is performed. Traditionally, models at different levels such as mentioned above require different CAD tools for the construction, simulation and analysis of the design. As a result, the designers generate several different models of the system (at different levels of detail) which do not interact with each other. The development of multiple disjoint representations of a common system under design results in the model continuity problem [3]. Models that have been developed and analyzed are often discarded once the analysis is completed and are not revisited in the reminder of the development process. This paper presents a technique called *hybrid modeling* which assists in bridging the gap between the level of performance modeling and lower levels of design by providing the ability to incrementally evolve a performance model into a behavioral design.

Performance modeling is a common approach for evaluating the performance of the system under design. These models are abstract in nature and their purpose is to estimate temporal performance metrics such as latency and utilization. Typically, performance models are based on queueing theory or Petri-Nets. The use of performance models in order to analyze the system performance in the earliest possible stage of the design process is being gradually adopted by the industry [4][5][6]. On the other hand, a full behavioral design includes all functional and temporal description of the design. This is the description of circuits at the implementation level and such description are referred to as interpreted model, i.e the value of system variables are defined for all times[7]. In an abstract performance model this is not the case and they are referred to as uninterpreted models.

Hybrid modeling, as the name implies, is a multi-level modeling approach, in which uninterpreted and interpreted elements can co-exist in a single model and its general structure is shown in Figure 1. Therefore, providing the capability of simulating abstract performance constructs and behavioral elements in a single simulation environment is the heart of this approach. Conceptually, hybrid modeling supports the stepwise refinement of abstract performance (uninterpreted) models into behavioral (interpreted) models.

The ability to support the evolution of a performance model into a behavioral model in a continuous fashion, using a top-down design process, is of great interest to many design environments [8][9].

Lately, more and more design teams have adopted the use of performance

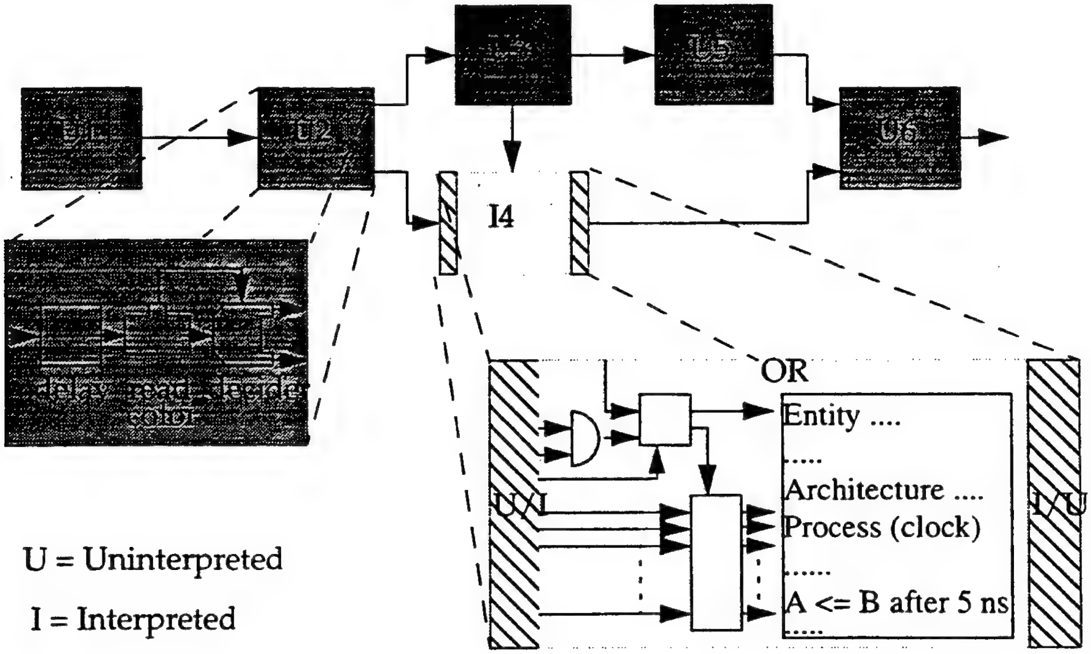


Figure 1: A hybrid model - General structure

modeling in the early stages of the design process, in order to examine different possible architectures and to estimate performance metrics for each case before commitment to a final architecture is made. This supports the process of defining an appropriate architecture for the system, and eliminates most architectural modifications often necessary at the integration step. It also supports the process of rapid prototyping of digital systems [10][11][12][13]. However, implementation errors, and in particular, implementation of subsystems in such a way that could result in insufficient performance, can still be detected only during the integration process, after the design of the entire system is completed and simulated. For example, if a particular subsystem was designed in a short time compare to the rest of the system, or if its design already exist (off-the-shelf component), its performance in the context of the system can still be verified only in the integration process. This situation implies that between the simulation of the performance model (that was constructed mainly for supporting architectural decisions) and the simulation of the fully implemented system, no intermediate stages of performance estimations currently exist.

The hybrid modeling approach is motivated by the need to fill the large gap between these two analysis environments and to enable performance verification during subsystem design. Ultimately, the result is that any subsystem that was designed at the behavioral level can be integrated into the latest existing performance model of the system, re-analyzed, and verified in the context of the model. This incremental process means that a major portion of the traditional integration stage is performed gradually over the complete design process. Therefore, the design modifications that will be required due to problems discovered in the integration process and the number of "last minute changes" will be reduced significantly.

The development of the hybrid modeling capability is motivated by other considerations as well. In most design processes, a combination of top-down and bottom-up design styles is employed [14][15]. In most cases, low-level implementation details have an impact on the system architecture. This situation is typically the case when the system is designed around an existing component or subsystem. Thus, design data must be freely transferred in both directions, top-down and bottom-up. The development of a model continuity capability between the performance and the behavioral domains of modeling will provide this feature. To go even further, the hybrid modeling capability can be used to examine different options of improving the performance of an existing system in a rapid manner. This will be done by integrating the performance model of a subsystem that is to be improved (or added) with the behavioral description of the existing system in a single simulation environment.

The motivation for hybrid modeling is strengthened by the so called risk-driven design approach. One such aspired design methodology was presented recently [9] and it is summarized in Figure 2. In this methodology, the major spiral cycles correspond to the iterations of a virtual prototype associated with the overall system. A virtual prototype is an executable model of the system and the stimuli that describe the system's operation at

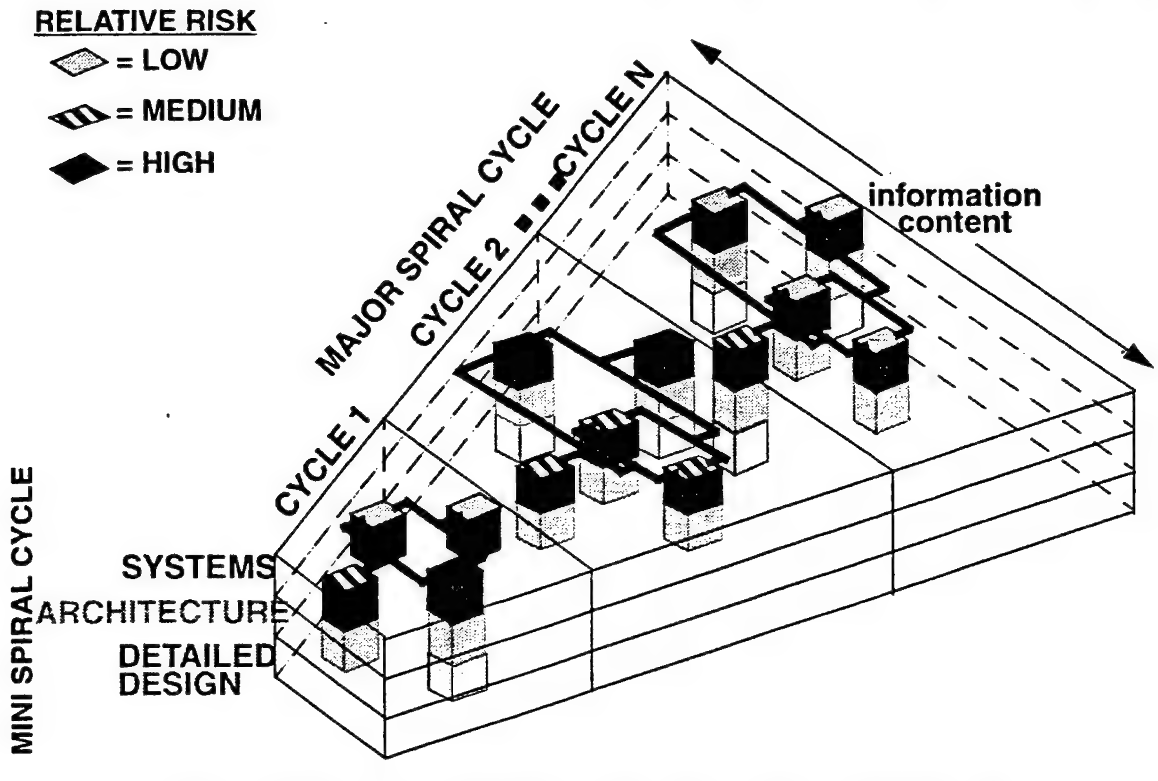


Figure 2: Risk Driven Expanding Information Model (RDEIM) [9]

different abstraction levels. The mini spiral cycles correspond to the iterations of a virtual prototype associated with portions of the design and/or models. This view corresponds to the notion that, based upon risk, pieces of the overall design may be at different levels of maturity. For example, in the first major cycle of Figure 2 the element with the highest relative risk is fully implemented (detailed design level) while the other elements are described at more abstract levels (system level or architectural level). If the simulation of the model shown in the first major cycle detects that the system will not meet its performance requirements, then the "risky" processing element is replaced by two similar elements operating in parallel. This result is shown in the second major cycle and at this point, another element of the system may become the new "bottleneck", i.e. the highest relative risk. As the design proceeds along the major spiral cycles, more information is included in the models. Hence the name "Risk Driven Expansion Information Model" (RDEIM) is used.

In order to support this design methodology, it is necessary to be able to simulate a model in which different elements are described at different level of abstraction. The hybrid modeling capability, which was the objective of this work, can provide the solution for that. This solution is especially true for systems where some elements are described at the detailed design level while others are described at an abstract (e.g. architectural) level in the same model.

In the design process, systems are usually partitioned into subsystems in such a way that most subsystems or components are sequential elements, i.e., these elements are activated by a clock signal and maintain state information. Such elements are referred to as synchronous sequential elements. This proposed research concentrates on the subset of hybrid models in which the interpreted element is a synchronous sequential element.

## 2 Hybrid Modeling

In typical uninterpreted (performance) models, tokens are used to represent the flow of information (data and control). When a hybrid model is to be developed from an initial performance model, the interpreted element that is to be included in the model has actual signals as inputs and outputs. The major challenge of developing a hybrid model is to resolve the difference in design detail that naturally exist at the interface between uninterpreted and interpreted elements. We refer to this interface as the *hybrid element interface*. Conceptually, the hybrid element interface is divided into two primary dependent parts: the part that handles the transition from the uninterpreted domain to the interpreted domain (U/I) and the part that handles the transition from the interpreted domain to the uninterpreted domain (I/U). These interfacing operations are affected by several attributes of the model and the system being modeled. In order to partition the hybrid modeling space and better define the specific solutions, a taxonomy was developed as the first step of this research effort [18].

## 3.1 Hybrid Modeling Taxonomy

The techniques for developing hybrid models depend on the class of problems being solved. The classes of hybrid modeling are defined by those model attributes which fundamentally alter the development and the implementation of the hybrid element interface. The hybrid modeling space is partitioned according to three major characteristics:

1) the evaluation *objective* of the hybrid model,

2) the *timing mechanism* of the uninterpreted model, and

3) the *nature* of the interpreted element.

For a given hybrid model, these three characteristics can be viewed as attributes of the hybrid model and the analysis effort. In this section, these three characteristics will be explained in more detail, and the reason for partitioning the hybrid modeling space along these lines will be clarified. Figure 3 summarizes the considerations discussed in this section.

**Hybrid Modeling Objectives:** The structure and the functionality of the hybrid element interface are strongly influenced by the modeler's objective. As explained earlier, the overall objective of hybrid modeling is to support true stepwise refinement from a performance model down to an implementation. This objective can be divided into two secondary objectives:

1. *Performance analysis* and *timing verification*:

    To analyze the performance of the system when one or more components are modeled in the interpreted domain, and to verify by simulation, that the interpreted component does not violate system timing constraints.

2. *Functional verification*:

    To verify by simulation that the function of the interpreted component (input-to-output values mapping) is acceptable, within the context of the system model.

Performance analysis of a hybrid model, which results in a more realistic performance estimation, is affected by the hybrid element in two different ways. The first way is due to the fact that the delay through the interpreted component itself is more realistic than the delay specified in the corresponding uninterpreted domain. This difference in delay affects the performance estimation of the system. The second way in which a hybrid element can affect system performance is due to some dependency of the

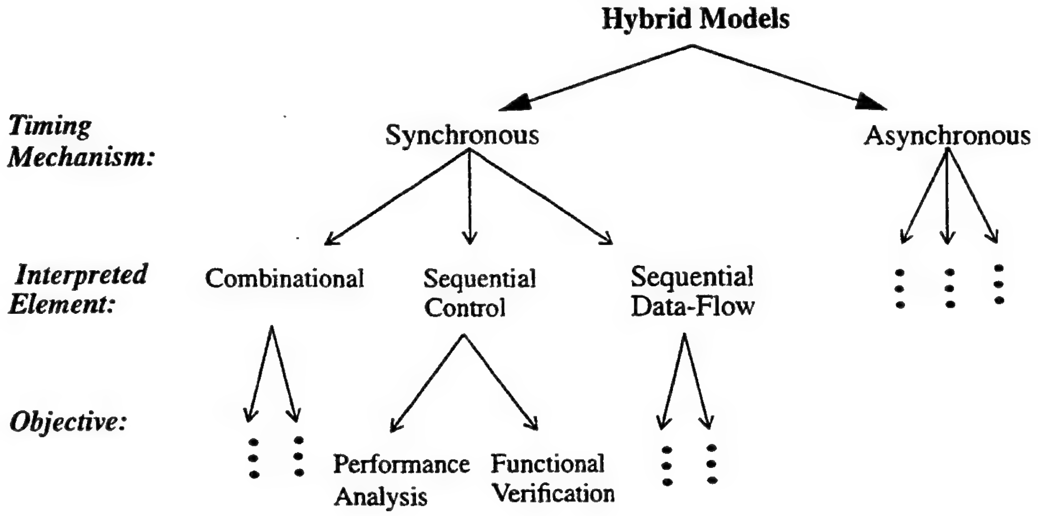


Figure 3: Hybrid Modeling categories

rest of the uninterpreted model on the interpreted component output values. The interpreted component contains full functionality and, the values on its output signals may be used to alter the token flow through the model. Therefore, the performance analysis objective and the functional verification objective are related.

Achieving both secondary objectives by using a single interface and a single technique is practical only if all input values to the interpreted component are known from the information within the tokens arriving from the uninterpreted model to the interface. For the case of unknown input values, i.e., the information on the tokens is not sufficient for determining all values at all times, functional verification may be very limited and will require a different technique from the one used for performance analysis and timing verification.

If the objective of the hybrid model is performance analysis only, the interface must detect when the interpreted element processing is completed, and release tokens to the rest of the model at the appropriate time. On the other hand, if the objective is functional verification only, the interface operates on output values from the interpreted domain rather than timing. It has to be emphasized that this functional verification objective is only in the context of the performance model, which inherently does not include all system functionality.

**Timing Mechanisms:** Typically, performance (uninterpreted) models are asynchronous in nature and the flow of tokens depends on the handshaking protocol. However, it is possible that during the model refinement, a decision to synchronize the flow of tokens will be made. The hybrid modeling technique depends upon this timing mechanism of the uninterpreted model. To be more specific, the synchronization at the hybrid interface will require different hybrid modeling approaches. However, since analysis is to be performed by sequentially performing hybrid modeling with different components in the system, it is necessary to reference the synchronization characteristics to the system model and not only at the interface. The two types of system models are:

1. *Asynchronous models*:

   Tokens on independent signal paths within the model move asynchronously with respect to each other.

2. *Synchronous models*:

   The flow of tokens in the model is synchronized by some global mechanism.

Synchronous models usually refer to models of systems with a single global clock, i.e., the global clock synchronizes all operations within the system, and the model of such a system reflects this synchronization scheme. An example of an asynchronous performance model is a model of a self-timed system. In other words, different parts of the systems are unclocked or operate with different clocks, or systems constructed of subsystems that communicate in an asynchronous fashion.

Constructing a performance model in an asynchronous fashion is more straight forward due to the delay-based nature of performance models. Constructing a synchronous model requires that some additional synchronization mechanism be added to the model. This synchronization can be done explicitly, e.g., by introducing a control-token corresponding to the clock which controls the data flow in the model. Another approach for introducing synchronization within the model is through implicit techniques that guarantee movement of tokens between components at certain times.

The functionality of the interface depends on the timing mechanism, especially in the case of multiple input token paths to the U/I operator. Since the U/I operator of the interface activates the interpreted element, multiple input token paths may be treated in several manners. For example, tokens may have to arrive at all input signals in order for activation, or, the first token that arrives may activate the interpreted block. Hence, the interfacing technique is strongly influenced by the synchronization of the model.

**Interpreted Component:** The hybrid modeling technique strongly depends upon the type of the interpreted component that is introduced into the performance model. It is natural to partition interpreted hardware descriptions into combinational elements and sequential elements. However, this research has suggested the following partition:

1. *Combinational Elements*:

Unclocked (with no states) elements, e.g., constructed of gates only.

2. *Sequential Control Elements* (SCE):

Clocked elements (with states) that are used for controlling data flow, e.g., a control unit or a controller

3. *Sequential Dataflow Elements* (SDE):

Elements that include datapath elements **and** clocked elements that control the data flow, e.g., control unit and datapath.

For combinational interpreted elements, the outputs depend on the current inputs only; therefore, the interface acts independently for each input token. On the other hand, for sequential interpreted elements, the interface must account for states, as well as inputs.

The major reason for partitioning the sequential elements into sequential dataflow and sequential control elements is based on the timing attributes of these elements. A sequential control element (SCE) is found in cycle-based machines, i.e., control input values are read every cycle and control output values (that control a datapath) are generated every cycle. On the other hand, sequential dataflow elements (SDE) have data inputs and may have some control inputs but the output data is usually generated several clock cycles later. This difference in the timing attribute will dictate a different technique for hybrid modeling.

This distinction can be further explained if we consider a digital system, such as an embedded system which moves through a sequence of states, to have associated with it information of two kinds: **data** which is being processed and **control** which manipulates the processing. If the interpreted element in a hybrid model is the control unit only (SCE), input and output control signals need to be represented in the original performance model. On the other hand, if the interpreted element includes both the control unit and the datapath (SDE), it is not necessary for the performance model to explicitly contain the control signals that control the datapath. In addition, if an SCE hybrid model is constructed, the objective of timing verification may include checks for satisfaction of timing specifications such as set-up and hold time. When an SDE hybrid model is constructed, timing verification and performance analysis are expressed in terms of number of cycles.

## 3.2 The Interfacing Operation

The hybrid modeling technique requires an interfacing operation that resolves the differences in design detail between the domains of interpretation. It is the interface that actually handles all interactions between the interpreted and uninterpreted elements. In a top-down design process, the interpreted element being introduced to the model replaces an uninterpreted part of the model. Therefore, the U/I operator has to supply values and timing for the input signals of the interpreted element according to the tokens arriving from the uninterpreted domain to the interface. Similarly, the I/U operator has to determine timing and values information for output tokens based on the output signals of the interpreted element. If the performance model includes all the information required for driving the input signals of the interpreted element then the interfacing operation is fully deterministic. In such cases, the U/I operator derives signal values (bits, integer etc.) that match the data types of the input ports of the interpreted element by reading the information from the input tokens (information is embedded in the color fields of the tokens). Similarly, the I/U operator must "bind" the interpreted element outputs to tokens, i.e., the outputs of the interpreted element can be used for adding or updating information in the color fields of the tokens before releasing them back to the uninterpreted domain.

However, in most cases, the performance model does not contain all the information required by the interpreted element. This situation is especially true for performance models used in a top-down design process, and also for performance models used for rapid prototyping purposes. Typically, the more abstract the performance model, the less information is contained in the color fields of the tokens. Therefore, the behavior of the hybrid interface in abstract models is not fully deterministic. We refer to this as the "unknown inputs" case.

The technique and methodology for hybrid modeling with combinational interpreted elements has been developed for both cases of known and unknown inputs, and is summarized in [16][17]. For the known inputs case, the challenge was to determine when all the outputs of the interpreted element have been stabilized and then to release a token accordingly. To solve this problem, a technique called 'time-expansion' was developed, in which fast-time domain and slow-time domain exist simultaneously in the hybrid model. For the unknown inputs case, a technique for determining the worst-case delay for a given confidence level was developed. The penalty in simulation time increases as the required confidence increases.

## 3.3 Interfacing Scenarios

Once all the attributes described above have been determined, several interfacing scenarios are possible. The interfacing scenarios define the dataflow across domains of interpretation. Intuitively, there seems to be two interfacing scenarios: data flowing from the Uninterpreted domain to the Interpreted domain (U/I) and data flowing from the Interpreted domain to the Uninterpreted domain (I/U). However, there is a third scenario which is best explained by an example. Consider a hybrid model in which an interpreted element replaces an uninterpreted element in the "middle" of an uninterpreted model, i.e., data flows from an uninterpreted domain to an interpreted domain and then back to an uninterpreted domain (Figure 4). In order to preserve token information, a third interfacing scenario, called U/I/U, is introduced. Therefore, whenever the interpreted block is surrounded by uninterpreted elements, it is regarded as a U/I/U interfacing scenario. The I/U interfacing is still required for the case of replacing a source of tokens, in the

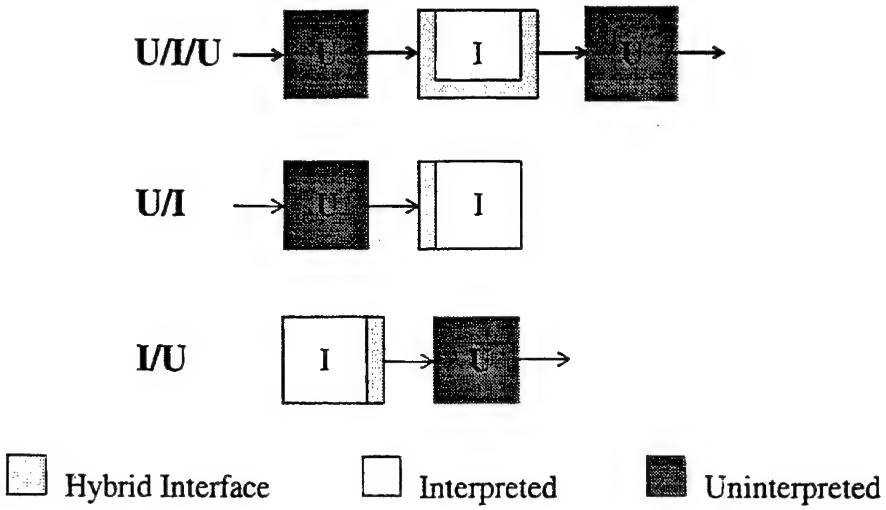


Figure 5: Interfacing scenarios

uninterpreted domain, with the behavioral description of the element that generates the data. Similarly, in the case of replacing a sink with the actual element, the U/I interfacing is to be used.

In general, the interfacing scenarios are determined by observing the surroundings of the uninterpreted block being replaced by an interpreted element. Therefore, only three interfacing scenarios are included in the hybrid modeling methodology: U/I/U, U/I, and I/U. (see Figure 5).

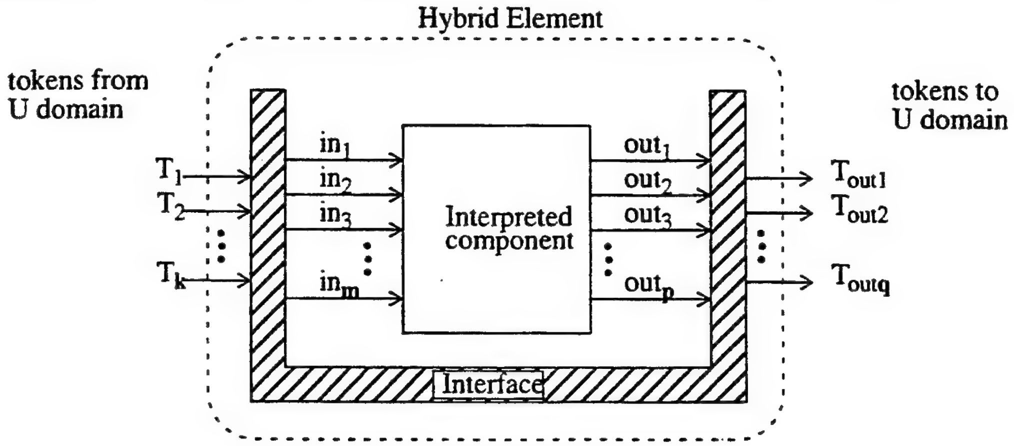


Figure 4: Signal Correspondence Between Modeling Domains

# 4 Hybrid Modeling with Sequential Interpreted Elements

This section is focused on describing the capability of hybrid modeling in which the interpreted element is an SDE (Sequential Dataflow Element) and the simulation objective is performance analysis. The focus on Sequential Dataflow Elements as interpreted elements is based on the fact that computer-based or data processing systems are most likely to be divided into such units during the design process. In the literature, such units are referred to as FSMD (Finite State Machines with a Datapath) [19]. They consist of a Finite State Machine (FSM) used as a control unit, and a datapath, as shown in Figure 6. Since the objective of uninterpreted models is usually to estimate the system performance in terms of throughput of data, systems are naturally partitioned to blocks which adhere to the FSMD structure. Each of these blocks (FSMDs) process the data and has some processing delays associated with it. Moreover, such blocks indicate the completion of a processing task to the rest of the system by some means of control outputs which are generated by the control unit embedded in the FSMD block. This feature of indicating the task completion is a key property to the methodology of constructing hybrid models with sequential interpreted elements.

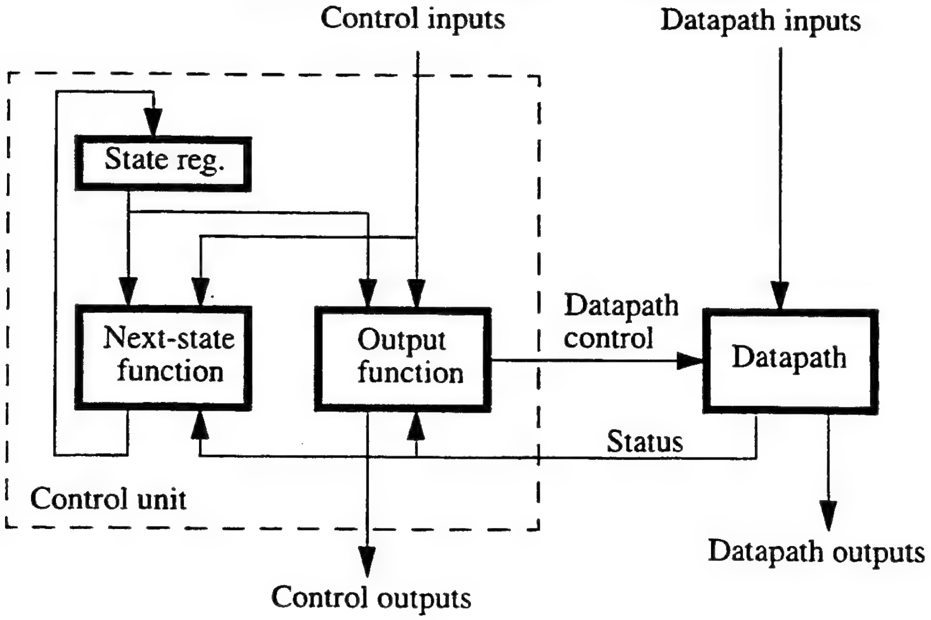


Figure 6: Generic FSMD block diagram

This methodology distinguishes between two cases, based on the approach for handling the lack of information between the different levels of abstractions. In the first case, the information required for stimulating the interpreted element is embedded within the abstract model and/or supplied by the modeler. We will refer to this case as the "known inputs" case. In the second case, it is assumed that there is an inherent information gap between the abstract (uninterpreted) part of the model and the interpreted element. We will refer to this case as the "unknown inputs" case, which means that at least some information which is required in order to stimulate the interpreted element is missing. When the hybrid modeling technique is used for the refinement of a performance model during the design phase of a system, it is expected that the lack of information between the levels of abstraction will be significant. In such cases, the hybrid modeling technique for the "unknown inputs" case will be applicable. Only towards later stages of the refinement process, when more information is embedded into the hybrid model, the "known inputs" case may become applicable.

## 4.1 The "Known Inputs" Case

This section describes the conceptual structure of the hybrid element, when the interpreted element is SDE and all its inputs are known. The same hybrid element structure is used for the "unknown inputs" case as well, except that the functionality of the U/I operator is expanded. This structure is independent of the design tool being used and it provides a general method for hybrid modeling. However, the incorporation of hybrid modeling into ADEPT adheres to this structure.

As explained earlier, the U/I operator has to drive the input signals to the interpreted element according to information carried within the input tokens, and/or information supplied externally by the modeler. The I/U operator has to release tokens at the appropriate timing and, potentially, with new values according to the output signals of the interpreted element. The structure of these two parts of the hybrid interface is shown in

Figure 7. The U/I operator is composed of the following building blocks: Driver, Activator and Clock-Generator. The I/U operator is composed of an Output_Condition_Detector, a Colorer and a Sequential_Releaser.

In the U/I operator, the Activator is used to signify the arrival of a new token (packet of data) to the interpreted element. This process is accomplished by using the Activator to drive control input/s of the interpreted element, as well as indicating the driver of a new token arrival. Its output is also connected to the I/U operator for gathering information on the delay through the interpreted element and for statistical analysis purposes. The Driver is used to “strip” information from the token’s color fields and to drive the datapath input signals (and potentially some control inputs as well) to the interpreted element according to predefined assignments properties. The Clock_Generator generates the clock signal in two possible modes: a free-run mode and a token-related mode. Usually, in abstract performance models, the clock signal is not expressed in the model. Therefore, the option of a free-run clock is necessary. In some cases, and especially for less abstract performance models, the clock signal may be expressed explicitly or implicitly in the model. In these cases, the Clock_Generator, which operates

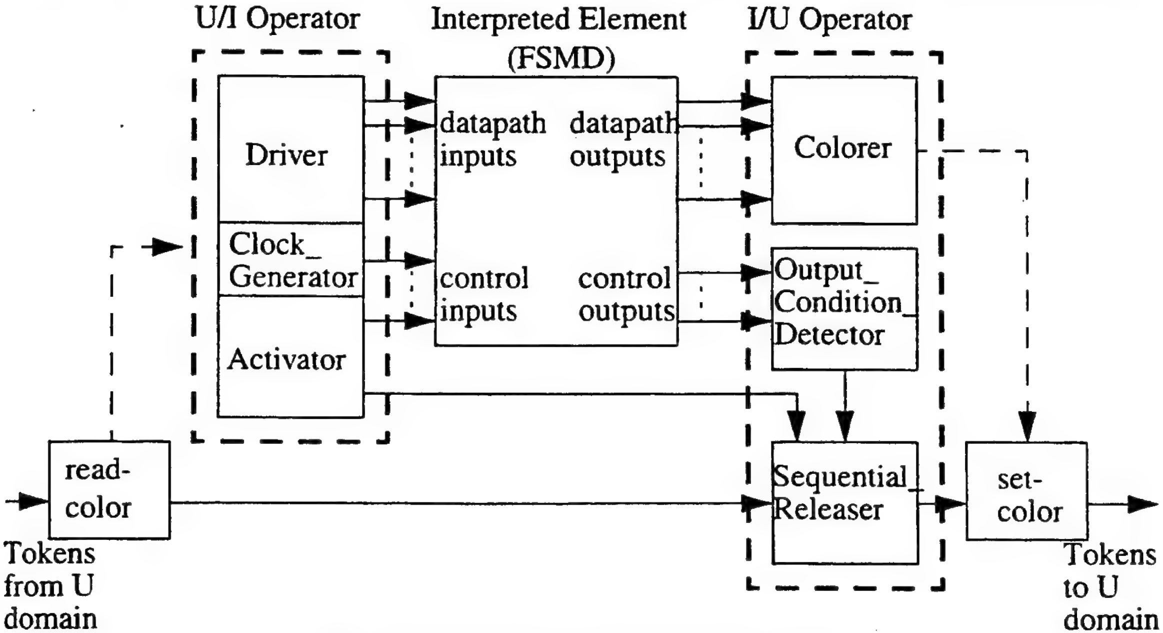


Figure 7: The Hybrid Element Structure

in a token-related mode, has to extract this information and to generate the clock signal accordingly.

In the I/U operator, the Output_Condition_Detector is used to signify the completion of the interpreted element data processing operation, by comparing the control outputs to predefined properties. This process is based on the typical feature of an FSMD, which indicates its data processing completion by asserting some output signals. The Colorer samples the datapath outputs and map them to color fields according to predefined binding properties. The Sequential_Releaser, which "holds" the original token, releases it back to the uninterpreted model upon receiving the signal from the Output_Condition_Detector. The information carried by the token is then updated by the Colorer and the token flows back to the uninterpreted part of the model.

A set of modules that support the construction of such hybrid interface has been implemented and added to the ADEPT modules library.

Given this structure, the operation of the hybrid element within the content of a hybrid model can be described. Upon arrival of a new token to the hybrid interface (U/I operator), the Activator notifies the driver to start its operation on a new packet of data. In the case of a Clock_Generator which works in the token-related mode of operation, the Activator will notify it to start generating a clock signal. Since the interpreted element is a sequential machine, the driver may need to drive sequences of inputs combination. In the "known inputs" case, which is described here, this information must be supplied to the driver, either by the token itself or by an external file. The driver supports both modes of operation. This sequence of inputs combination is supplied to the interpreted element, while the original token is held by the Sequential_Releaser. This token is released back to the uninterpreted model only when the Output_Condition_Detector indicates the completion of the interpreted element operation. This operation of token releasing may take a varying number of clock cycles, depending on the information carried with the token, as well as the status of the interpreted element.

In this "known inputs" case, the interpreted element performs its computation on deterministic data. Therefore, it is useful to utilize this hybrid model not only for performance analysis but also for functional verification, within the context of the hybrid model. The Colorer is used to map the output data of the interpreted element onto color fields of the token, which is released back to the uninterpreted model. The new information which is carried by the token, may be used by the uninterpreted model, e.g. for routing decisions. Therefore, examining the simulation results of the hybrid model can provide some means of validating the output values generated by the interpreted element. This verification may be limited and it is not suppose to replace the "stand-alone" functional verification of the interpreted element. However, it can help verifying the appropriate use of the interpreted element in the system being modeled.

## 4.2 The "Unknown Inputs" Case

The hybrid modeling technique provides the capability of dealing with unknown inputs which result from the abstract nature of a performance model. Typically, the more abstract the performance model, the higher the ratio of unknown to total inputs. In some cases, particularly during the very early stages of the design process, it is possible that the abstract performance model will not provide any information to the interpreted element, other than the fact that a new data has arrived. If some (or all) inputs are not known from the token, some criterion for value selection must be made. Choosing a criterion is based on the objective of the hybrid model. For the objective of "performance analysis and timing verification", delays (number of clock cycles) through the interpreted element are of interest. The most common criterion in such cases is the worst-case processing delay. In some cases, best-case delay may be desired. If the number of unknown inputs is small, exhaustive search for worst/best case may be practical. Therefore, it is desired to minimize the number of unknown inputs which can affect the delay through the interpreted element. The methods for achieving this objective are described conceptually in section 4.2.1. By utilizing these methods, the number of unknown inputs is likely to be reduced but unknown inputs will not be eliminated completely. In this case, the performance metrics of best and worst case delay can be provided by some means of a "traversal" process. Section 4.2.2 describes this developed traversal method for determining the delays through a sequential element and clarifies its usefulness for hybrid modeling. Formal algorithms are also described.

### 4.2.1 Reducing the Number of Unknown Inputs

Reducing the number of unknown inputs can simplify the simulation of a hybrid model significantly. Although, utilizing the following methods is not essential for achieving the objective of "more realistic performance estimation", it is included as part of the hybrid modeling methodology.

One way of reducing the number of unknown inputs is by an ad-hoc approach which utilizes the knowledge provided by the behavioral description of the interpreted element, such as the "meaning" of some input signals. However, a more algorithmic approach was developed. Since FSMD elements have output signals that signify "the completion of data processing", other outputs are not significant for performance analysis. Therefore, the "non-significant" (insignificant) outputs are considered as "don't care". By projecting them back to the inputs, it is possible to minimize the number of unknown Delay Affecting Inputs (DAIs). The term "projecting back" implies that if an input signal affects <u>only</u> the values of insignificant outputs, then this signal is not a DAI. Therefore, projecting insignificant outputs to don't-care inputs is equivalent to determining those inputs which do not affect the values on the significant (in terms of performance) outputs. The major steps in this algorithm are:

<u>Step 1</u>: Select the "insignificant" outputs (in terms of temporal performance).

<u>Step 2</u>: In the STG, replace all values for these outputs with a "don't-care" (called: the modified state machine).

**Step 3**: Minimize the modified state machine (generate the corresponding state table).

**Step 4**: Find the inputs which do NOT alter the flow in the modified state machine (by detecting identical columns in the state table, and combining them by implicit input enumeration).

This method is best illustrated by an example. Consider the state machine which is represented by the state transition graph shown in Figure 5-1. This simple example is a state machine with two inputs, $X_1$ and $X_2$, and two outputs, $Y_1$ and $Y_2$. This machine cannot be reduced; i.e., it is a minimal state machine. Assume that this machine is the control unit of an FSMD block and that the control output $Y_1$ is the output which indicates the completion of the "data processing" operation (when its value is 1) while output $Y_2$ is an "insignificant output"(e.g. used as a control signal that animates the datapath). Therefore, a "don't-care" value is assigned to $Y_2$ and according to step 2, the modified STG is shown in Figure 5-2. Based on step 3 of the method, an attempt to minimize this modified state machine is performed. In this example, states A, C and E are equivalent (can be replaced by a single state, K) and the minimal machine consists of three states, K, B and D. This minimal machine is described by table Table 5-1.

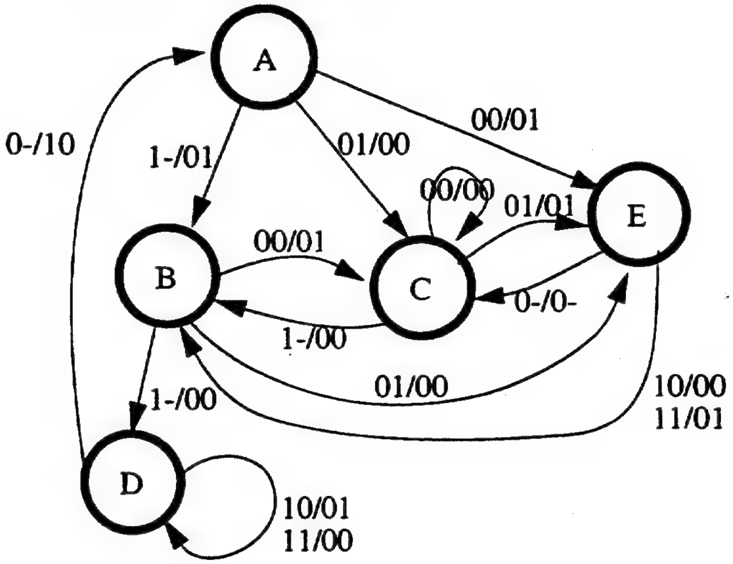


Figure 5-1 : STG of a 2-inputs 2-outputs state machine

**Table 5-1: Next-State and output $Y_1$ of the minimal machine**

| P.S. \ $X_1X_2$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| K=(ACE) | K, 0 | K, 0 | B, 0 | B, 0 |
| B | K, 0 | K, 0 | D, 0 | D, 0 |
| D | K, 1 | K, 1 | D, 0 | D, 0 |

All possible input combinations appear explicitly in Table 5-1. However, it can be seen that the first two columns of the table are identical (i.e. the same next state and output value for all possible present states). Similarly, the last two columns of the table are identical. This observation can lead to a more compact table that represent the same minimal machine but with implicit enumeration of the inputs. This is shown in Table 5-2.

**Table 5-2: The minimal machine with implicit input enumeration**

| P.S. \ $X_1X_2$ | 0- | 1- |
|---|---|---|
| K=(ACE) | K, 0 | B, 0 |
| B | K, 0 | D, 0 |
| D | K, 1 | D, 0 |

This table shows that the minimal machine does not depend on the value of input

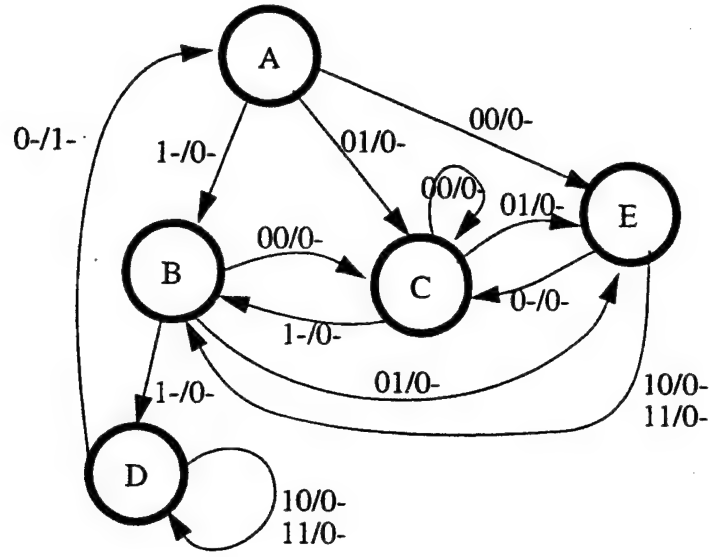


Figure 5-2 : STG with “significant” output values only

$X_2$. Therefore, the conclusion is that input $X_2$ is definitely not a Delay Affecting Input. This result implies that by knowing the value of input $X_1$ only, the number of clock cycles (transitions in the original STG) required to reach the condition that output $Y_1 = 1$ can be determined, regardless of the values of $X_2$. This is the case for any given initial state. It is important to emphasize that the paths in the original STG and their lengths are those which must be considered. It is also important to remember that the modified state machine is used only for the purpose of detecting non-DAIs since it was generated by assigning "don't-care" value to the "insignificant" outputs (step 2). The machine which is actually being traversed during the hybrid simulation is the original state machine with all its functionality.

To demonstrate the meaning of an input which is not a DAI, consider the original state machine represented by the graph in Figure 5-1 and assume that the initial state is A. Consider, for example, one possible sequence of values on input $X_1$ to be 0, 1, 0, 0, 1, 1, 0. By applying this input sequence, the sequence of values on output $Y_1$ is 0, 0, 0, 0, 0, 0, 1, regardless of the values applied to input $X_2$. Therefore, two input sequences which differ only in the values of the non-DAI input $X_2$ will produce the same sequence of values on the "significant" output $Y_1$. For example, the sequence $X_1X_2$ = 00, 10, 00, 01, 10, 10, 01 will drive the machine from state A to E, B, C, E, B, D, and back to A, and the sequence of values on output $Y_1$ will be as above. Another input sequence, $X_1X_2$ = 01, 11, 01, 00, 11, 11, 00, in which the values of $X_1$ are identical to the pervious sequence, will drive the machine from state A to C, B, E, C, B, D, and back to A, while the sequence of values on output $Y_1$ is identical to the previous case. Therefore, the two input sequences will drive the machine via different states but will produce an identical sequence of values on the "significant" output (which also implies that the two paths have an equal length).

Although one may see why the method presented above will detect only non-DAIs, a more formal explanation is presented here. Generally, a synchronous sequential machine

M is a quintuple $M = (I, O, S, \delta, \lambda)$ where I, O, and S are finite non-empty sets of inputs outputs and states, respectively [23][24].

$\delta : I \times S \rightarrow S$ is the state transition function;

$\lambda$ is the output function such that

$\lambda : I \times S \rightarrow O$ for Mealy machines [25];

$\lambda : S \rightarrow O$ for Moore machines [26].

Since Mealy machine represent the more general case, the rest of this section will discuss this case. The method for finding inputs which are not DAIs is based on partitioning the set of outputs, O, to two disjoint subsets, $O^S$ and $O^N$. The elements of the subset $O^S$ are the "significant" outputs (in terms of performance) while the elements of the subset $O^N$ are the "insignificant" outputs. This partition is assumed to be done by the designer, based on the knowledge of the output's functionality. Based on this partition, the objective of the method presented previously is to partition the inputs set to two disjoint subsets, $I^D$ and $I^N$. The elements of the subset $I^N$ are those inputs which are definitely non-Delay Affecting Inputs while the elements of the set $I^D$ are inputs which may affect the delay, i.e. the inputs which affect the values of the outputs in the set $O^S$. Therefore, the output function $\lambda$ is also partitioned to two mapping operations such that

$$\lambda_1 : I^D \times S \rightarrow (O^S \cup O^N)$$

$$\lambda_2 : I^N \times S \rightarrow O^N$$

The output function $\lambda_2$ implies that the inputs which are elements of the subset $I^N$ can affect the values of the outputs which belong to the subset $O^N$ only.

The next step is to show that, for a given partition on the set O (to $O^S$ and $O^N$), the method described previously will produce a partition on the input set I, based on the output functions $\lambda_1$ and $\lambda_2$.

Lemma: All the inputs that are constantly "don't-care" in the minimal machine with implicit input enumeration (generated by step 4 of the method) are elements of the set $I^N$.

Proof: Assume that an input $i_n \in I$ was detected by the method to be a non-DAI, i.e., $i_n \in I^N$. Suppose that this input is actually a Delay Affecting Input, i.e., it should be an element of the set $I^D$ instead of the set $I^N$. Therefore, based on the output functions $\lambda_1$ and $\lambda_2$, there is at least one output $o_k \in O^S$ that its value is a function of the input $i_n$ for at least one given state $s_m \in S$, i.e. $o_k = f(I^D \cup i_n, S)$. This implies that, if $s_m$ is the present state, $o_k(I^D \cup (i_n = 0), s_m) \neq o_k(I^D \cup (i_n = 1), s_m)$. In such a case, the minimal state machine (generated in step 3 of the method) will be represented by a state table in which at least two columns have a different value for $i_n$ (in their input combination headers). These two columns will have at least one row (the one that corresponds to $s_m$ being the present state) in which the value of the output $o_k$ is different. Therefore, these two columns cannot be combined into a single column in the minimal state table with implicit input enumeration (generated in step 4 of the method). This situation means that the input $i_n$ cannot have a "don't-care" value in all columns headers of this state table. Therefore, $i_n \notin I^N$, which contradicts the initial assumption.

111

### 4.2.2 Traversing the STG for Best and Worst Delay

After extracting all possibilities for minimizing the number of unknown inputs, a method for determining values for those inputs which remain unknown is required. This method is based on the traversal of the STG of the finite state machine embedded within the FSMD element. As noted in Figure 6, this state machine is part of the control unit within the SDE interpreted element. The STG traversal should provide the best/worst case delay in terms of number of clock cycles (which is equivalent to the number of transitions in the STG). As explained earlier, some combination of control output values may signify the completion of processing the data. The search algorithm will look for such control output combination. The justification for this approach is that the significant event, from performance analysis perspective, is the release of a token back to the uninterpreted model. Therefore, the search algorithm will look for maximum/minimum number of transitions from a given initial state to a certain output combination. Since the state machine is represented by a State Transition Graph which is a directed graph, this search process is equivalent to finding the longest/shortest path in the STG.

A state in the FSM is represented by a node in the STG. Therefore, a given initial state is mapped to an initial node in the STG. A transition in the FSM is represented by an arc in the STG. In a general Mealy state machine, the output values are attached to transitions. Therefore, a given output combination is mapped to an arc, or several arcs in the STG. This situation means that a path from the initial node to one of these arcs is searched for.

The search for the shortest-path utilizes a well-known algorithm. Search algorithms exist for both, single-source shortest-path and all-pairs shortest-path. One of the first and most commonly used algorithm is Dijkstra's algorithm [27], which finds the shortest-path from a specified node to any other node in the graph. The search for all-pairs shortest-path is also a well investigated problem. One such algorithm by Floyd [28] is based on work by Warshall [29]. Its computation complexity is $O(n^3)$ when n is the number of nodes in the graph, which makes it quite practical for moderate-sized graphs. The implementation of this algorithm is based on Boolean matrix multiplication and the actual realization of all-pairs shortest-paths can be stored in an $n \times n$ matrix. Utilizing this algorithm required some enhancements in order to make it applicable for hybrid modeling. For example, if some of the inputs to the interpreted element are known (from the token), then the path should include transitions that do not contradict these known inputs.

On the other hand, the search for the longest-path is a more complex task. It is an NP-complete problem and has not attracted significant attention. Since most digraphs contain cycles, they need to be handled during the search in order to prevent from a path to contain a cycle infinite number of times. One possible restriction is to construct a path that will not include a node more than once. Given a digraph *G(V,E)* which consists of a set of vertices (or nodes) $V = \{v_1, v_2 \ldots\}$ and a set of edges (or arcs) $E = \{e_1, e_2 \ldots\}$, a **simple-path** between two vertices $v_{init}$ and $v_{fin}$ is a sequence of alternating vertices and edges $P = v_{init}, e_n, v_m, e_{n+1}, v_{m+1}, e_{n+2}, \ldots, v_{fin}$ in which each vertex does not appear more than once.

Given an initial node and a final node, the algorithm starts from the initial node and adds nodes to the path in a Depth-First-Search (DFS) fashion, until the final node is reached. At this point, the algorithm backtracks and continues looking for a longer path. However, since the digraph may be cyclic, the algorithm must avoid the possibility of increasing the path due to a repeated cycle, which may produce an infinite path.

The underlying approach for avoiding repeated cycles in the algorithm dynamically eliminates the cycles while searching for the longest-simple-path. Let *u* be the node that the algorithm just added to the path. All the in-arcs to node *u* can be eliminated at this stage of the path construction; i.e., setting the in-degree of *u* to be zero. The justification for this dynamic modification of the graph is that, while continuing in this path, the simple-path cannot include *u* again. While searching forward, more nodes are being added to the path and more arcs can be removed temporarily from the graph. At this stage, two things may happen: 1) either the last node being added to the path is the final node or, 2) the last node has zero out-degree in the dynamically modified graph. These two cases are treated in the same way except that in the first case the new path is checked to see if it is longer than the longest one found so far. If it is, the longest path is updated. However, in both cases the algorithm needs to backtrack.

Backtracking is performed by removing the last node from the path, hence decreasing the path length by one. During the process of backtracking, the in-arcs to a node being removed from the path must be returned to the current set of arcs. This process will enable the algorithm to add this node when constructing a new path. At the same time, whenever a node is removed from the path, the arc that was used in order to reach that node is marked in the dynamic graph. This process will eliminate the possibility that the algorithm repeats a path that was already traversed. Therefore, by dynamically eliminating and returning arcs from/to the graph we can treat a cyclic directed graph as if it does not contain cycles. The process of reconnecting nodes, i.e. arcs being returned to the dynamic graph, requires that the original graph be maintained. The exact algorithm can be found in [20].

The STG is usually a cyclic directed graph which includes nodes with multiple in-arcs and out-arcs. Therefore, a more realistic restriction on the longest-path is that it will not include any arc more than once. Such a longest-path with no repeated arcs may include a node multiple times as long as it is reached via different arcs. In the case of more than one transition that meets the condition on the output combination, a search for the longest-path should check all paths between the initial state and all these transitions. However, such a path should include any of these transitions only once, and it should be the last one in the path.

The hybrid modeling methodology, which is composed of all the methods described above, has been added to the UM methodology and integrated into ADEPT. The steps for minimizing the unknown inputs can be performed prior to simulating the hybrid model. On the other hand, the search for longest/shortest possible delay should be performed during the simulation itself. This situation is due to the fact that each token may carry different information which may alter the known input values and, therefore, alter the search of the STG. The STG traversal was integrated into the design environment utilizing the following steps: 1) when a token arrives to the hybrid interface, the simulation is halted and the search for minimum/maximum number of transitions is performed, and 2) upon completion of the search, the simulation continues while applying the sequence of inputs found in the search operation. The transfer of information between the VHDL simulator and the search program (C-code) is done by using the simulator interface.

Since hybrid models are part of the design process and are constructed by refining a performance model, it is likely that many tokens will carry identical relevant information. This information may be used for selective application of the STG search algorithm, hence increasing the efficiency of the hybrid model simulation. For example, if several tokens carry exactly the same information (and assuming the same initial state of the FSM), the search is performed only once, and the results can be used for the following identical tokens.

## 4.3 Validation

Based on the explanation given so far, one should see the advantage of hybrid modeling as part of the design process. However, the purpose of this section is to validate this advantage in a more formal way. In other words, a formal answer is needed to the following question: Why and how the technique of hybrid modeling with sequential interpreted elements will reduce the "risk" in the design process?

Reducing the risk in the design process can be achieved by using a design methodology that will provide quick and useful *feedback* for any *design decision*.

In this case:

- The *feedback* is how the system performance is affected by the delay (latency) through a sequential interpreted element, when incorporated into the performance model of the entire system (particularly the upper and lower bounds on this delay).
- The *design decision* is the way this interpreted element was chosen to be implemented.

In the uninterpreted performance model, there is delay associated with the part that represents the sequential element. This delay may be a fixed value but it can also be a data-dependent delay. In both cases, this delay is set by the modeler and is based on certain assumptions. Therefore, this delay is considered as an estimated delay $\hat{D}$ of the true delay $D$. The term "true delay" stands for the actual delay through the interpreted element after it has been implemented.

When the delay associated with the element in the uninterpreted model is a fixed delay then $\hat{D} = constant$. When the delay associated with the element in the uninterpreted model is a data-dependent delay then $\hat{D} = f(tag_1, tag2, ..., tag15)$, or more generally, $\hat{D}$ is a function of the information carried by the input token.

For a given sequential interpreted element, the true delay $D$ is a function of the values on its inputs. Some of these inputs, $\{k_1, k_2, ..., k_n\}$, may be derived from the input token ("known inputs") and some are unknown, $\{u_1, u_2, ..., u_m\}$. Therefore,

$$D = h(KnownInputs, UnknownInputs) = h(k_1, k_2, ..., k_n, u_1, u_2, ..., u_m)$$ , while

$$\hat{D} = f(KnownInputs) = f(k_1, k_2, \dots, k_n).$$

It can be shown that in the general case $\hat{D}$ is a **biased estimator** i.e. $E[\hat{D}] \neq D$, which implies that the estimated delay does not represent the true delay adequately. This process represents the result of fitting an inadequate model for the delay [21][22].

Consider the case of a data-dependent delay in the uninterpreted model. This situation is equivalent to fitting the model $D = K \cdot \Theta + \varepsilon$ and for the arrival of a single token (i.e. a single experiment): $D = \begin{bmatrix} k_1 & k_2 & \dots & k_n \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \\ \dots \\ \theta_n \end{bmatrix} + \varepsilon$

Where $K$ is the set of values for the known inputs, $\Theta$ is a set of parameters for the given element, the delay $D$ is the response and $\varepsilon$ is the estimation model error.

If the delay of the actual component really depends on the "known inputs" only (i.e. on the data carried by the token) then the above model is adequate and the modeler has to assign the correct values for the parameters $\begin{bmatrix} \theta_1 & \theta_2 & \dots & \theta_n \end{bmatrix}$. However since the performance model is usually abstract, the delay depends on some of the "unknown inputs" as well. This situation is assumed to be the general case and the above model is not adequate for it. The model which should have been fitted is $D = K\Theta_1 + U\Theta_2 + \varepsilon$, where

$K = \begin{bmatrix} k_1 & k_2 & \dots & k_n \end{bmatrix}$ are the "known inputs"

$U = \begin{bmatrix} u_1 & u_2 & \dots & u_m \end{bmatrix}$ are the "unknown inputs"

$\Theta_1 = \begin{bmatrix} \theta_{11} \\ \theta_{12} \\ \dots \\ \theta_{1n} \end{bmatrix}$ is one set of parameters

$\Theta_2 = \begin{bmatrix} \theta_{21} \\ \theta_{22} \\ \dots \\ \theta_{2m} \end{bmatrix}$ is another set of parameters

While fitting an inadequate model, the response is considered as a biased estimation of the true response. The inadequacy of the model with respect to the correct model is represented by the *bias* (or *alias*) *matrix* which is defined as [22]:

$$A = (K^T \cdot K)^{-1} \cdot K^T \cdot U$$

The only case in which the estimator $\hat{D}$ is unbiased, i.e. $E[\hat{D}] = D$, is when $A = \mathbf{0}$. Now, $A = \mathbf{0}$ only if $K^T \cdot U = \mathbf{0}$ which is possible only if $K$ is orthogonal to $U$ or if $U = \mathbf{0}$ (which is also the condition for orthogonality when only one experiment is performed).

$U = \mathbf{0}$ means that the delay does not depend on the "unknown inputs" and is a function of the "known inputs" only. As explained earlier, this is a special case in which the original model is an adequate one.

So far the case of data-dependent delay in the uninterpreted model has been considered and it has been shown that the estimated delay $\hat{D}$ is biased. If the adequate model is the one with two sets of parameters $\Theta_1$ and $\Theta_2$ then the case of fixed delay in the uninterpreted model is certainly a biased estimator.

The hybrid modeling technique developed in this work requires an access to the State Transition Graph (STG) of the sequential element. By traversing this STG, the effect of the "unknown inputs" on the delay (number of clock cycles) through the element can be found. This process provides the possible true delays through the interpreted element since its implementation is fully behavioral.

To find the effect of the "unknown inputs" on the delay, all possible combinations for values of $\{u_1, u_2, \dots, u_m\}$ should be considered. This means that an exhaustive search of the STG is required. However, in most cases, not a complete factorial experiment is required (factorial experiment implies $2^m$ experiments). There are two reasons for this:

I. If some of the inputs are known, only those transitions with input values that do not contradict the given values of the "known inputs" are considered during the traversal process.

II. Most practical STGs are "partial-state-graphs" which means that input values associated with a transition may also be "don't-care". Figure 6 is an example.

Therefore, the required search will not be as computation intensive as a full factorial experiment and it can still find all possible delays (number of transitions) through the sequential element.

When the objective of the hybrid model is performance and timing analysis, practical search results are the upper and lower bounds on the delay. Therefore, search algorithms for the shortest and longest possible paths (in terms of number of transitions) in the STG are used. They will find the true maximum and minimum delay through a given interpreted element and for given values for the "known inputs". The true delay in the extreme cases, $D_{max}$ and $D_{min}$, provide much more realistic information than the biased estimation of the delay $\hat{D}$ in the uninterpreted model. Therefore, by detecting the bounds on the true delay before the entire system is implemented, the risk of not meeting the performance requirements is reduced.

This chapter started with describing the techniques and the conceptual solutions to hybrid modeling for both known and unknown inputs cases. The following chapter elaborates on the methods for determining the delay through the sequential interpreted

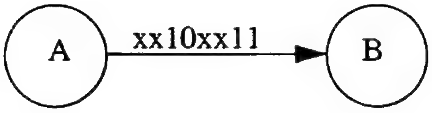


Figure 6An example of a transition in a "partial-state-graph" element in the “unknown inputs” case. The algorithms being developed to support these methods are presented in detail.

## 4.4 An Example

This example demonstrates the construction of a hybrid model as part of the design process. A performance model of the system under design is first constructed followed by the replacement of a portion of this performance model with its behavioral level detailed implementation.

A performance model of an execution unit was created. This execution unit is composed of an Integer Unit (IU), a Floating-Point Unit (FPU) and a Load-Store Unit (LSU). These units operate independently although they receive instructions from the same queue (buffer of instructions). If the FPU is busy processing one instruction, the following instruction which require the FPU is buffered, waiting for the FPU to be free again. Meanwhile, instructions which require the IU can be consumed and processed by IU at an independent rate. Both the FPU and the IU have the capability of buffering only one instruction. Therefore, if two or more consecutive instructions are waiting for the same unit to be free, the other units cannot receive new instructions (since the second instruction is held in the main queue).

A performance model for the system was constructed in ADEPT. The purpose of this model is to estimate the performance of this execution unit for different possible traces of instructions. One practical performance metric is the time required for the execution unit to process a given sequence of instructions.

It is assumed that this performance model was constructed at the beginning of a design process, and is a part of a larger performance model for the system (microprocessor) under design. One possible architecture is modeled for performance estimation. The delays in this model are based on estimated processing time for different types of instructions. At the beginning of the design process, when no complete traces exist, the performance model is based on some statistical assumptions. For example, it is estimated that the Integer Unit will require 30 ns for most integer instructions such as Add, Subtract etc. but will require much longer time (150 ns) for the Division operation. Also, since no complete traces exist at this stage, it is assumed that a certain percentage of the instructions will include the Division operation. This kind of assumption strongly depends on the algorithm that will be executed by this execution unit. However, modifying these values can be done by modifying a single parameter in the model, which is the threshold level of a random generator. It is expected that the performance model constructed at these early stages of the design process will be abstract and based on statistical decisions. However, it is also expected that this model will be mutated to include more information, hence eliminating statistical decisions, as the design advances.

Now, lets assume that the Floating-Point Unit has been designed to the behavioral level, or that an existing design of the FPU is considered to be used as an “off-the-shelf” block. The option of waiting until all parts of the system will be designed, and only then integrating them always exist. However, it is often desirable to examine whether utilizing an existing FPU in this system will provide the desired performance for the system. A hybrid model, in which the detailed description of the FPU is introduced into the performance model, may provide the answer to this question.

The hybrid model is constructed according to the methods described earlier and by using the ADEPT modules created for hybrid models with sequential interpreted element. Since the performance model is abstract while the FPU is described at the behavioral level, the lack of information at the hybrid interface is inevitable at early stages of the design. This situation will require employing the methods for “unknown inputs” which were described earlier. By simulating this model, an upper and lower bound on performance are obtained. As more information is provided by the traces, the number of unknown inputs to the interpreted element will be reduced and the range limited by the lower and upper bound will get smaller.

The hybrid model is shown in Figure 7. The hybrid interface is constructed around the interpreted block which is the behavioral description of the FPU. This FPU is an FSMD type of element. Since the FPU is assumed to be previously designed, the finite-state-machine which describes the control unit is assumed to be provided, as well as the VHDL description of this control unit. The inputs to this state machine indicate the operation to be performed (Add, Sub, Comp, Mul, MulAdd and Div), the precision of the operation (single or double) and some other control information. The number of clock cycles requires to complete any instruction depends on these inputs.

The first step, which is performed prior to the simulation, is to find if there are any inputs which do not affect the latency through the FPU. It is given that only one output indicates that the data processing has been completed. Therefore, the other outputs are considered to be a “non-significant” outputs in terms of performance. With this information given, the method described in Section 4.2.1 is executed by running the program on the given STT, and projecting the “non-significant” outputs to “don’t-care” inputs. Non-Delay Affecting Inputs were detected, which implies that if the other inputs to the control unit are supplied by the abstract model, e.g. as information carried by the tokens, then the upper bound and lower bound of the delay through the FPU will be identical. In other words, the performance estimation obtained from a simulation under these conditions is a single value, and will be identical to the performance estimation when all the interpreted inputs are known, regardless of the value assigned to the non-Delay Affecting Inputs. This kind of information is very useful to the modeler, since it tells which inputs are required to be known in order to simulate precise delays through the interpreted element.

Figure 8 shows the execution unit performance for three different traces. The ordinate axis is normalized by defining a unity to be the amount of time required to process a trace according to the uninterpreted performance model. The other metrics, upper and lower bounds on the performance, are all relative to the performance obtained

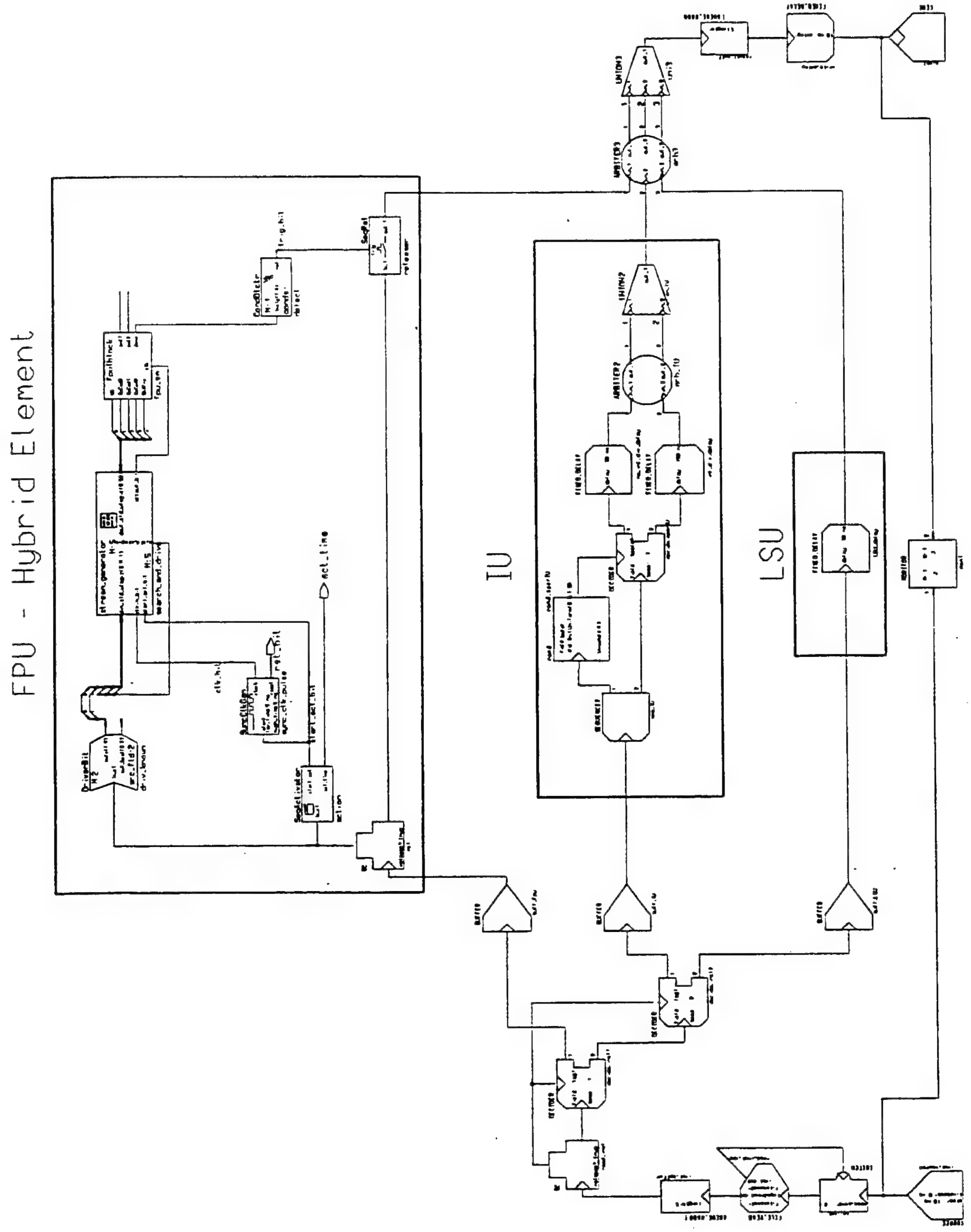


Figure 7: Hybrid model with the FPU behavioral description

by the uninterpreted model. For the simulation results shown in the graph, only 40% of the inputs were known from the token. It is clear that for some traces the range between the upper and lower bounds can be very wide while for other cases it can be relatively small. The benefit of the simulation results of the hybrid model is clear: not just a performance estimation based on a statistical uninterpreted model but actual performance limits for a given implementation of the FPU.

## 5 The Refinement Process

Although a hybrid model can provide performance estimation that may illuminate necessary architectural or implementation changes, this capability is only the first step in the comprehensive refinement process. At this point, there are two major ways of continuing this refinement process:

I. Abstraction reduction - make the uninterpreted part of the model less abstract.
II. Multiple hybrid elements - create more hybrid elements in the same model.

Simulation results of the initial hybrid model may generate an estimated

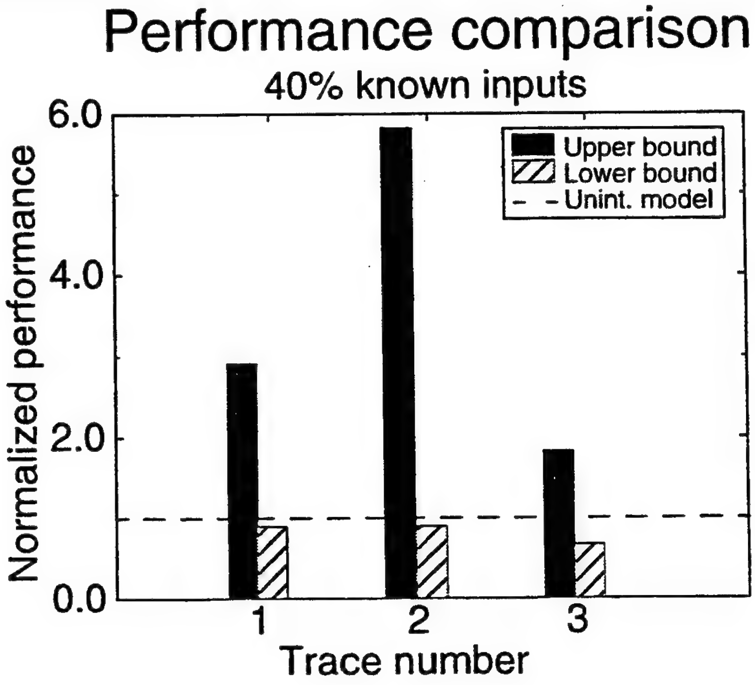


Figure 8: Performance comparison for three traces

performance range which is too wide for reaching operational conclusions. In such a case, the modeler is interested in decreasing the possible range of latencies. One way of achieving this goal is by reducing the number of "unknown inputs" to the interpreted element. Since the information required in order to drive more "known inputs" should be supplied by the uninterpreted part of the hybrid model, implications are that the level of abstraction of the system description should be reduced. Adding the information to the uninterpreted part of the model should not be done arbitrarily. It should be targeted towards providing information that can be used to drive the interpreted element. At this point, the method for detecting non-DAIs can be useful. If any inputs were detected as non-DAIs, the modeler should target his effort of abstraction reduction towards including information that can be useful for driving the other inputs, i.e. those who are potentially Delay Affecting Inputs. If the model reaches a situation in which enough information is included in order to drive all the inputs which are potentially DAIs, no more effort should be invested in reducing the model abstraction at this point. The simulation of this hybrid model will generate a sole performance metric, i.e. zero range.

As an example, the hybrid model of the execution unit was modified to include more information in the uninterpreted part. Figure 9 shows the performance obtained from simulating a single trace for different number of known inputs. Again, the ordinate axis is normalized according to the performance obtained from simulating the uninterpreted model. It can be seen that when none of the inputs to the interpreted element is known, the difference between the upper and lower bound is large. In this case, the search for the longest-path in the FPU will always produce the double-precision division operation (which takes 34 clock cycles) while the search for the shortest-path will always produce the sequence for one of the fast operations (such as addition).

As more information is provided by the token, more inputs are known, and, therefore, the range bounded between the best and worst case is getting smaller. It is interesting to observe the case in which 80% of the inputs are known. In this case, the upper and lower bounds are identical which means that the hybrid model simulation produced a sole performance metric. This result is expected to happen when all inputs are known. However, as explained earlier in this example, the first step was to "project" the non-significant outputs (in terms of performance) to the inputs. It was found that the value on some inputs do not affect the performance of the FPU state machine. This is the only unknown inputs in the case of 80% known inputs in the graph. This result demonstrates the advantage of detecting the non-DAIs prior to the hybrid model simulation, hence, informing the modeler which inputs are not necessary to be known without affecting the quality of the performance estimation.

Given an initial hybrid model with one interpreted element, the refinement process can also be performed in a different manner. Based on the risk-driven approach, the simulation results of the initial hybrid model may discover a new "risky" element (bottle neck) in the system. Such a scenario was explained along with Figure 1-1 (Risk Driven Expanding Information Model). In such a case, the modeler may choose to introduce the

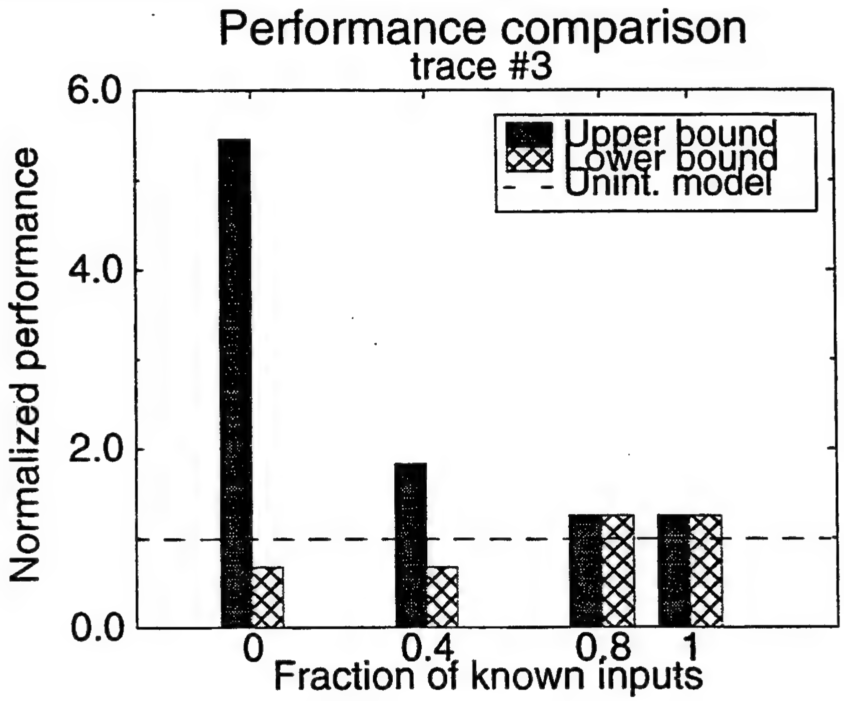


Figure 9: Performance vs. fraction of known inputs

detailed behavioral information of this "risky" element into the model, thus, creating a hybrid model with two separate hybrid elements.

The hybrid modeling technique that was incorporated into ADEPT provides the capability for multiple hybrid elements in a single model. It also supports multiple hybrid elements with "unknown inputs". Based on whether the maximum or minimum latencies through the interpreted elements are chosen, multiple search processes on different state machines are performed during the simulation. As in the "known inputs" case, the interfaces can be constructed by using the hybrid library modules, allowing the modeler to link between each interface and its corresponding interpreted element and state machine.

There is one practical drawback to this refinement approach. When the performance model is abstract, it is likely that the interpreted elements will have unknown inputs. This situation implies that the search for the extreme delays has to be performed on multiple machines. A search process, especially for the maximum delay, is a complex operation which requires intensive computation and it may slow down the simulation to an unacceptable speed. In order to overcome this problem, some modifications to the refinement process should be considered. While simulating the initial hybrid model it is possible to collect data on the latency of the interpreted element. Then, while introducing another interpreted element to the model, the first interpreted element can be removed and replaced by the corresponding uninterpreted description with the actual delays found in the first simulation. This method of back-annotation [30] can reduce the amount of time required in order to simulate the hybrid model without affecting the accuracy of the performance estimation.

# References

[1] R. Goering, "Designers reach for a higher level", *Electronic Engineering Times*, August 3, 1992.

[2] K. Keutzer, "Panel: The ESDA Landscape", *Proceedings 32nd Design Automation*

*Conference*, San Francisco, 1995, pp. 534.

[3] D. Franke, M. Purvis, "Hardware/Software Codesign: A Perspective", *13th International Conference on Software Engineering*, May 1991, pp. 344-352.

[4] A. Poursepanj, "The PowerPC Performance Modeling Methodology", *Communication of the ACM*, Vol. 37, No. 6, June 1994.

[5] M. Bekerman, A. Mendelson, "A Performance Analysis of Pentium Processor Systems", *IEEE Micro*, October 1995 pp. 72-83.

[6] M. Tremblay, G. Maturana, A. Inoue, L. Kohn "A Fast and Flexible Performance Simulator for Micro-Architecture Trade-off Analysis on UltraSparc-I", *Proceedings 32nd Design Automation Conference*, San Francisco, 1995, pp. 2-6

[7] J. H. Aylor, R. Waxman, B. W. Johnson, R. D. Williams, "The Integration of Performance and Functional Modeling in VHDL" in *Performance and Fault Modeling with VHDL*, J. Schoen Editor, Prentice-Hall Inc., 1992, pp. 22-144.

[8] C. Rose, "The What and How of Top Down System Design", TD Technologies, March 1993.

[9] Martin Marietta Laboratories, RASSP First Annual Interim Technical Report (CDRL A002), Moorestown, October 31, 1994.

[10] V. K. Madisetti, M. A. Richards, "Advances in Rapid Prototyping of Digital Systems", *IEEE Design and Test of Computers*, Fall 1996, pp. 9-11.

[11] V. Madisetti, "Rapid Digital System Prototyping: Current Practice, Future Challenges", *IEEE Design and Test of Computers*, Fall 1996, pp. 12-22.

[12] S. Banerjee, P. M. Chau, R. D. Fellman, "Rapid Prototyping Methodology for Multiprocessor Implementation of Digital Signal Processing Systems", *Journal of VLSI Signal Processing*, Vol. 11, No. 1-2 pp. 21-34.

[13] M. B. Srivastava, R. W. Brodersen, "SIERA: A Unified Framework for Rapid-Prototyping of System-Level Hardware and Software", *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 14, No. 6, June 1995, pp. 676-693.

[14] E. Shragowitz, H. Yousesef, L. Bening, "Predictive Tools in VLSI System Design: Timing Aspects", *COMPEURO - System Design: Concepts, Methods and Tools*, 1988, pp 48-55.

[15] M. C. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Description", *Proceedings 23rd Design Automation Conference*, 1986, pp. 474-480.

[16] MacDonald R., "Hybrid Modeling of Systems with Interpreted Combinational Elements", Ph.D. dissertation, University of Virginia, May 1995.

[17] MacDonald, R., Williams. R., Aylor, J., "An Approach to Unified Performance and Functional modeling of Complex Systems", *IASTED Conference on Modeling and Simulation*, April 1995.

[18] M. Meyassed, R. Mcgraw, J. Aylor, R. Klenke, R. Williams, F. Rose, J. Shackleton, "A Framework for the Development of Hybrid Models", *Proceedings 2nd Annual RASSP Conference*, Arlington, Va, July 1995.

[19] D. Gajski, N. Dutt, A. Wu, S. Lin, "HIGH-LEVEL SYNTHESIS: Introduction to Chip and System Design", Kluwer Academic Publishers, 1992.

[20] M. Meyassed, "System-Level Design: Hybrid Modeling with Sequential Interpreted Elements," Ph.D. Dissertation, Department of Electrical Engineering, University of Virginia, January, 1997.

[21] R. V. Hogg, J. Ledolter, "Applied Statistics for Engineers and Physical Scientists", MacMillian Publishing Co., New York, 1992.

[22] G. E. Box, N. R. Draper, "Empirical Model-Building and Response Surfaces", John Wiley & Sons, 1987.

[23] Z. Kohavi, "Switching and Finite Automata Theory", 2nd Ed. McGraw-Hill Inc. 1978.

[24] F. J. Hill, G. R. Peterson, "Introduction to Switching Theory and Logical Design", 2nd Ed., John Wiley & Sons, 1974.

[25] G. H. Mealy, "A Method for Synthesizing Sequential Circuits", *Bell Systems Tech. J.*, Vol. 34 pp. 1045-1079, Sep. 1955.

[26] E. F. Moore, "Gedanken-experiments on Sequential Machines", pp. 129-153, *Automata Studies, Annals of Mathematical Studies*, No. 34, Princeton University Press, Princeton, N. J., 1956.

[27] E. W. Dijkstra, "A note on two problems in connection with graphs", *Numerische Math. 1, 269-271 (1959).*

[28] R. Floyd, "Algorithm 97 (shortest path)", *Communications of the ACM 5(6), 345 (1962).*

[29] S. Warshall, "A theorem on boolean matrices", *Journal of the ACM 9(1), 11-12 (1962).*

[30] Z. Navabi, M. Massoumi, "Investigating simulation of hardware at various levels of abstraction and timing back-annotation of dataflow description", Simulation, Vol.57, No. 5, Nov. 1991, pp. 321-332.

# Mixed-Level Modeling in VHDL Using the Watch-and-React Interface

William W. Dungan, Robert H. Klenke, and James H. Aylor

Department of Electrical Engineering
University of Virginia
Charlottesville, VA 22903

## Abstract

*Using VHDL, it is possible to model systems at many different levels of detail. The various modeling levels (performance, behavioral, etc.) can also be intermixed to create mixed-level models. This paper describes the watch-and-react interface which was created to resolve the differences in timing and data abstraction between the performance modeling domain (token based) and the behavioral modeling domain (value based). Specifically, this interface is useful for integrating behavioral models of complex sequential components into performance models. It operates by monitoring the "important" signals in a system and then reacting to changes in these signals by generating tokens or forcing signals to appropriate values given the particular situation. The two main elements in the interface are the trigger and the driver. Program files containing scripting instructions are interpreted by the these two elements as the VHDL model simulates.*

## 1. Introduction

As a digital system designer, one would like to evaluate system design alternatives as early as possible in the design process in order to achieve higher quality designs with the lowest possible system development costs [1]. As different design alternatives are evaluated, some parts of the system may evolve more than others. Traditionally, many different development tools have been used to evaluate these design alternatives. There are tools for high level stochastic modeling [2], tools for critical path detection [3], and tools for functional verification. One disadvantage of a design process that uses several of these tools, is having to recreate a representation of the system in each tool. This adds costs to the development process that could have been avoided if a tool existed that provided a single-path, unified, design environment.

In the early stages of a design, performance models are often created. In performance models, the flow of information, not the form or value of that information, is important. At this level, models can be thought to pass *tokens* which are meant to represent information with its form and value abstracted away. This modeling level is termed uninterpreted since components cannot interpret information, but must act solely on the presence (or absence) of it as represented by tokens. Models made of these high-level components, which use the passing of tokens to represent the flow of information, are therefore called uninterpreted models.

Within the Center for Semicustom Integrated Systems (CSIS) at the University of Virginia, a hardware description language-based uninterpreted modeling tool called ADEPT (Advanced Design Environment Prototyping Tool) has been developed [4,5]. ADEPT is a unified design environment developed to support system evolution from concept to implementation. It uses VHDL as the common language for such a single-path development environment. With this VHDL-based design environment, it is possible to build performance models which can be incrementally refined to an actual implementation.

In the context of ADEPT performance models, a token is implemented as a VHDL record that is passed between modeling components via a four-state handshake protocol on interconnecting signals. Small amounts of integer-type information can be propagated through a model on token data fields (called tags). Throughput and latency are the typical performance metrics that can be estimated based on token flow and tag field data in ADEPT models.

In the later stages of a design, behavioral models for portions of the system are created that have significantly more detail than performance models, in which information has form and value. Unlike uninterpreted components, behavioral components contain functionality responsible for mapping values at their inputs to values at their outputs and typically contain more detailed timing and event granularity. A behavioral model can also be called an interpreted model if it contains only behavioral or interpreted components. As system components are refined to the interpreted level, it is very beneficial to simulate their models in the context of the entire system. In order to be able to perform this refinement in an incremental fashion, the capability to cosimulate uninterpreted and interpreted components in the same model is required. A so called "mixed-level interface" must be

---

**The authors would like to acknowledge the support provided by the Defense Advanced Research Projects Agency under contract number F33615-93-C-1313.**

**Appeared in the Proceedings of the Fall VHDL International Users Forum, 1997, pp. 25-32**

placed between the uninterpreted and interpreted components.

In behavioral models, signals are usually of a less abstract data type than tokens (such as bit, std_logic, integer, or real) and must have actual values associated with them in order for the model to function correctly. In addition, behavioral models usually resolve timing events to a finer granularity than uniterpreted models. This mixed-level modeling interface must therefore resolve these differences in timing and data abstraction between the performance modeling domain (token based) and the behavioral modeling domain (value based).

For example, consider the example of a performance model of multicomputer system in which one of the processors is replaced with an Instruction Set Architecture (ISA) level behavioral model. The performance model represents data packets passing between processors as abstract tokens that include the size of the data packet, but not the values it contains. In contrast, a single data packet (token) in the performance model may require the ISA level processor model to execute hundreds or thousands of simulated bus cycles (timing abstraction) and the ISA level processor model will require actual data values (data abstraction) to be placed on its data bus in order to simulate correctly. Obviously, the mixed-level interface must expand the single token into the required number of processor bus cycles and provide the necessary values on the processor data bus.

This paper presents a mixed-level modeling interface intended resolve the data and timing abstraction differences between performance models and complex sequential interpreted components such as microprocessors. This interface is called the "watch-and-react" interface. The remainder of this paper is organized as follows; Section 2 presents the details of the watch-and-react interface. This includes the components that constitute the interface and how they are used. Section 3 presents two modeling examples which demonstrate how the watch-and-react interface can be used to integrate behavioral components into performance models. Finally Section 4 presents some conclusions

## 2. Elements of the Watch-and-React Interface

A methodology and components for constructing a mixed-level interface involving general sequential interpreted components that can be described as Finite State Machines (FSMs) was detailed in [6]. However, many useful mixed-level models can be constructed that include sequential interpreted components that are too complex to be represented as FSMs, such as microprocessors, floating point coprocessors, etc. The watch-and-react interface was created to be a generalized, flexible interface between these types of interpreted components and ADEPT performance models.

Some of the terminology associated with hardware and software monitoring techniques has been adopted in describing the watch-and-react interface. The *state* of a system can be defined by the values contained in various storage elements such as flip-flops, registers, and memory locations [7]. The values in these storage elements are reflected by the values on various signals in the system. For example, the value stored in a specific memory location is reflected by the value on the memory bus when that memory location is accessed. In most performance measurement situations, only a few of these states are relevant. That is, only the relevant states tell anything interesting about the performance of the system. The signals that are important in defining relevant states are referred to as *primary variables*. Changes in primary variables are referred to as *events*.

The two main elements in the watch-and-react interface are the *trigger* and the *driver*. Figure 1 illustrates how the trigger and driver are used in a mixed-level interface. Both elements have ports that can connect to signals in the interpreted components of a model. Collectively, these ports are referred to as the *probe*. The primary job of the trigger is to detect events or changes in primary variables on its probe, while the primary job of the driver is to force values onto the signals attached to its probe. The trigger encodes information about events onto ADEPT tokens and the driver decodes information carried in ADEPT tokens to determine what values to force onto signals.

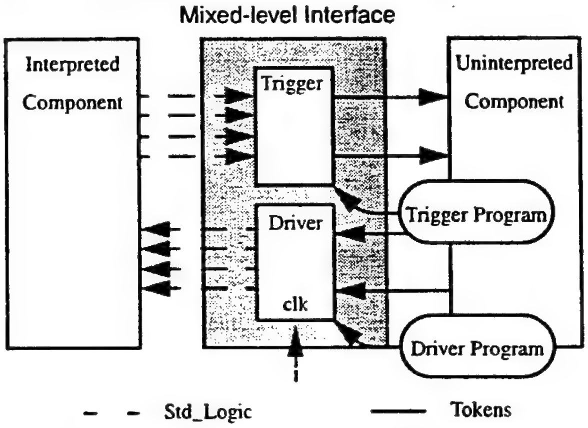


Figure 1: The watch-and-react interface

The trigger and driver are used in the watch-and-react interface to handle the translation of values to tokens and tokens to values. The trigger and driver were designed to be as generic as possible. It is largely the responsibility of the designer to specify how this translation is done using these components. An interface language has been designed that specifies how the trigger and driver should behave. This

**Appeared in the Proceedings of the Fall VHDL International Users Forum, 1997, pp. 25-32**

language is interpreted by the trigger and driver during the simulation and has syntax similar to a some constructs of VHDL.

## 2.1 Trigger

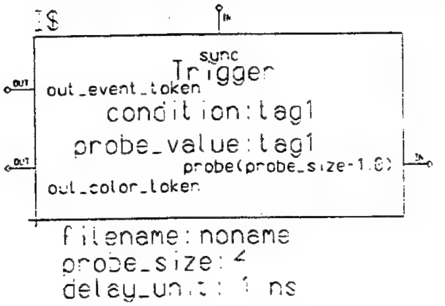


Figure 2: Schematic symbol for the trigger

The primary job of the trigger is to detect events or changes in primary variables. The schematic symbol for the trigger is shown in Figure 2. The probe on the trigger is a bus of std_logic signals `probe_size` bits wide, where `probe_size` is a generic on the symbol. There is one token output called `out_event_token` and one token output called `out_color_token`. Tokens generated by the trigger when events are detected are placed on the `out_event_token` port. The condition number (an integer) of the event that caused the token to be generated is placed on the `condition` tag field, which is specified as a generic on the symbol. Also, each time a signal changes on the probe, the color of the token on the `out_color_token` port changes appropriately regardless of whether an event was detected or not. The probe value is placed on the `probe_value` tag field of the `out_color_token` port. The `sync` port is used to synchronize the actions of the trigger element with the driver element as explained below.

The name of the file containing the trigger's program is specified by the `filename` generic on the symbol. The `delay_unit` generic on the symbol is a multiple that is used to resolve the actual length of an arbitrary number of delay units specified by some of the interface language statements.

## 2.2 Driver

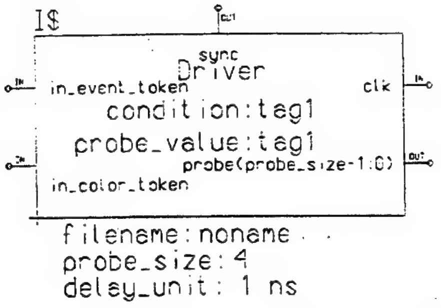


Figure 3: Schematic symbol for the driver

The primary job of the driver is to force values onto its probe. The schematic symbol for the driver element is shown in Figure 3. The probe on the driver is a bus of std_logic signals `probe_size` bits wide, where `probe_size` is a generic on the symbol. There is one token input called `in_event_token`, one token input called `in_color_token`, and a special input for a std_logic type clock signal called `clk`. The `clk` input allows driver to synchronize its actions with an external interpreted clock source. Under normal operating conditions, tokens are decoded as they arrive on the `in_event_token` port and the appropriate values are forced onto the probe. However, when the driver does dynamic driving the value on the `probe_value` tag field of the `in_color_token` is forced onto the probe. The `sync` port is used to synchronize the actions of the driver element with the trigger element.

The name of the file containing the driver's program is specified by the `filename` generic on the symbol. The `delay_unit` generic on the symbol is a multiple that is used to resolve the actual length of an arbitrary number of delay units.

## 2.3 Interface Language

The watch-and-react interface language is an interpreted language that has similarities to VHDL. It was created to allow users to easily describe the behavior of the trigger and driver elements. Because the language is interpreted, changes can be made to the trigger and driver programs without having to recompile the model. There are several guidelines that should be followed when writing programs for the trigger and driver.

1. The first statement in the program must be either `trigger` or `driver`. This statement denotes the element for which the program was written and thus determines how the program will be parsed.

2. The interface language is case sensitive. The inappropriate use of case will result in an unpredictable execution of the program.

3. All non-empty lines must begin with a line number. Line numbers should be unique and in ascending order. The line numbers are used to label statements so that control flow statements such as `goto` will work correctly.

4. Statements reserved for trigger programs should not be used in driver programs and vice versa.

### 2.3.1 Common Trigger and the Driver Statements

The statements discussed in this section are recognized by both the trigger and driver elements. They can be used in exactly the same way in program files for either of the two elements without their syntax or semantics changing.

`-- <comment>`

This statement is used to comment programs. Everything on the same line after the comment statement is ignored.

`alert_user`

This statement will generate a warning message in the VHDL simulator output window. It is intended to be used by designers to signify a specific point in either a trigger or driver program has been reached.

`delay_for <T>`

This statement produces a delay of T delay units, where T is an integer. The delay unit is specified as a generic on both the trigger and the driver.

`end`

This statement marks the end of a program. An `end` statement must appear as the last statement in every program even if the flow of execution will never reach it. The `end` statement can also be used at other places one wishes the program to terminate.

```
for <N>
    <loop- body>
next
```

This statement is used to iterate N times over the loop body, where N is an integer. The loop body is a sequence of statements. However, `for-next` statements cannot be nested.

`goto <L>`

This statement allows the flow of execution of a program's statements to continue with the statement at line number L, where L is an integer. If there is no matching line number, then an error occurs.

`output_sync`

This statement causes a token to be generated on the `sync` port of either the trigger or driver. It should be used in conjunction with the `wait_on_sync` statement, described below, to synchronize trigger and driver elements.

`wait_on_sync`

This statement halts the flow of execution until a token arrives on the `sync` port of either the trigger or driver.

### 2.3.2 Trigger Specific Statements

The statements discussed in this section are specific to the trigger. They can be used in conjunction with the statements discussed in Section 2.3.1, but should not be used in driver programs.

```
case_probe_is
when <STD_LOGIC_VAL>
    <sequence of statements>
when...
end_case
```

This statement is used to conditionally execute some sequence of statements associated with the probe taking on the value STD_LOGIC_VAL, where STD_LOGIC_VAL is a std_logic_vector. Once a `when` statement in a `case` clause evaluates to true, the flow of execution continues with the sequence of statements associated with that `when` statement and subsequently with the statements following the `end_case`. A `case` clause can have any number of `when` statements and the sequence of statements associates with each `when` statement can be any number of statements in length.

`output <INTEGER_VAL> after <T>`

This statement generates a token with the value INTEGER_VAL on the tag field specified by the condition tag field generic, where INTEGER_VAL is an integer. The token will become present after T basic delay units, where T an integer.

`trigger`

This statement must appear as the first statement in a program for the trigger element.

`wait_on <STD_LOGIC_VAL>`

This statement halts the flow of execution until the probe takes on the value STD_LOGIC_VAL, where STD_LOGIC_VAL is a std_logic_vector.

`wait_on_probe`

This statement halts the flow of execution until there is a signal event on the probe.

### 2.3.3 Driver Specific Statements

The statements discussed in this section are specific to the driver. They can be used in conjunction with the statements discussed in Section 2.3.1, but should not be used in trigger programs.

```
case_token_is
when <INTEGER_VAL>
    <sequence of statements>
when...
end_case
```

This statement is used to conditionally execute some sequence of statements associated with the probe taking on the value INTEGER_VAL, where INTEGER_VAL is an integer. Once a `when` statement in a `case` clause evaluates to true, the flow of execution continues with the sequence of statements associated with that `when` statement and subsequently with the statements following the `end_case`. A `case` clause can have any number of `when` statements and the sequence of statements associates with each `when` statement can be any number of statements in length.

**`driver`**

This statement must appear as the first statement in a program for the driver element.

**`dynamic_output_after`** `<T>`

This statement forces values onto the probe after T basic delay units. The forced values comes from the `probe_value` tag field of the token present on the `in_color_token` port. T is an integer.

**`output`** `<STD_LOGIC_VAL>` **`after`** `<T>`

This statement forces the value STD_LOGIC_VAL onto the probe after T basic delay units. STD_LOGIC_VAL is a std_logic_vector and T is an integer.

**`wait_on`** `<INTEGER_VAL>`

This statement halts the flow of execution until a token arrives with the value INTEGER_VAL on the tag field specified by the `condition` tag field generic, where INTEGER_VAL is an integer.

**`wait_on_fclk`**

This statement halts the flow of execution until the falling edge of the clock is detected.

**`wait_on_rclk`**

This statement halts the flow of execution until the rising edge of the clock is detected.

**`wait_on_token`**

This statement halts the flow of execution until a token arrives on the `in_event_token` signal.

## 3. Modeling Examples

Two modeling examples are presented in this paper that demonstrate how to use the watch-and-react interface for mixed-level modeling. The first example is of a counter which counts from zero to eight and then resets. This model demonstrates the basics of how to use the trigger and driver. The second example is of a digital control system. In this system there is an interpreted model of a microprocessor-based controller and an uninterpreted model of a motor controller system.

### 3.1 Counter Example

The purpose of this simple example is to demonstrate how the trigger and driver can be used to recognize and react to signals in a mixed-level model. There are two interpreted elements (a clock and a four-bit counter), along with one trigger and one driver. There are no explicit uninterpreted elements; the token output from the trigger connects directly to the token input of the driver. The counter has two inputs; one for a reset signal and one for the clock. The reset is active low and the counter increments on the rising clock edge. Should the counter ever reach "1111", it will wrap around to "0000" and continue counting.

The probe of the trigger is connected to the counter's output, the probe of the driver is connected to the counter's reset input, and tokens generated by the trigger as events are detected are fed into the driver. The clock is connected to the counter and driver's clock inputs. A schematic of the model is shown in Figure 4.

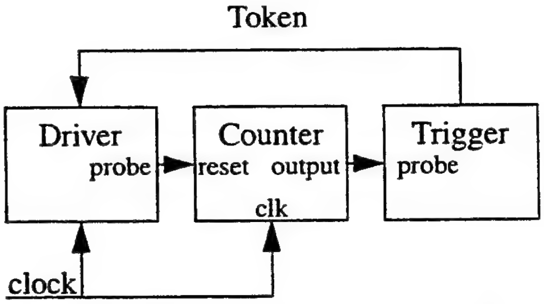


Figure 4: Schematic of counter model

For this example the trigger is programmed to recognize when the probe value is either "0000" or "1000". When the probe value is "0000", the trigger outputs a token instructing the driver to force the reset line high. When the reset line is high, the counter operates normally. When the probe value is "1000", the trigger outputs a token instructing the driver to force the reset line low. This action causes the counter to reset to "0000". This process effectively causes the counter to count from "0000" to "1000" continuously.

The program for the trigger is listed in Figure 5. The first statement in the file identifies it as a trigger program. The program begins on line 10 by waiting for a change to occur on the probe. Once a change is detected, the `case` statement is evaluated. If there is a `when` statement which matches the probe value, then the statements associated with that `when` statement will be executed. Afterwards, the flow of execution will continue with the statements following the `end_case`. If none of the `when` statements match, then the flow of execution will continue with the statements following the `end_case` without executing any of the statements associated with the `when` statements. In both cases, the program will branch back to line 10, where the program will wait for a change to occur on the probe.

```
trigger
10 wait_on_probe
20 case_probe_is
30     when 0000
40         -- normal operating condition
50         output 1 after 0
60     when 1000
70         -- reset condition
80         output 2 after 0
90 end_case
100 goto 10
110 end
```

Figure 5: The trigger program file for the counter model

The program for the driver is listed in Figure 6. There are many similarities between the trigger program and the driver program in this example. This first statement in the file identifies it as a driver program. The program begins on line 10 by waiting for a token to arrive. Once a token arrives, the `case` statement is evaluated. If there is a `when` statement which matches the value of the token's `condition` tag field, then the statements associated with that `when` statement will be executed. Afterwards, the flow of execution will continue with the statements following the `end_case`. If none of the `when` statements match, then the flow of execution will continue with the statements following the `end_case` without executing any of the statements associated with the `when` statements. In both cases, the program will branch back to line 10, where the program waits for a change to occur on the probe.

```
driver
10 wait_on_token
20 case_token_is
30     when 1
40         -- reset <= 1 after 5 ns
50         output 1 after 5
60     when 2
70         -- reset <= 0 after 5 ns
80         output 0 after 5
90 end_case
100 goto 10
110 end
```

Figure 6: The driver program file for the counter model

As an example, consider the case when the output of the counter changes to "1000". The signal trace for this example is shown in Figure 7. Assuming that the trigger's program is halted on line 10, this condition will enable the flow of execution to continue. Line 60 of the trigger's program matches this probe value, and the output statement on line 80 generates a token with the value of 2 on its `condition` tag field. When it arrives at the driver (as shown by the "present-acked-released-removed" cycle on the token-status field), this token will wake up the driver's program which is halted on line 10 waiting for a token to arrive. Line 60 of the driver's program matches with the token's condition tag field value of 2, and the `output` statement on line 80 forces the reset line to '0' after 5 delay units, causing the counter to reset to "0000". Assuming that the `delay_unit` generic on the driver is set to 1 nanosecond, this delay will be resolved as 5 nanoseconds. The driver's program will then continue after the `end_case` and jump to line 10, where it will wait for another token to arrive from the trigger. The trigger will generate its next token when the counter changes to "0000", in which case a cycle similar to the one described here will result in the driver changing the reset line to '1' after which the counter will begin counting normally.

```
time (ns)  clk   reset   output   token-status
   100      1      1      1000        removed
   100      1      1      1000        present
   100      1      1      1000          acked
   100      1      1      1000       released
   100      1      1      1000        removed
   105      1      0      1000        removed
   105      1      0      0000        present
   105      1      0      0000          acked
   105      1      0      0000       released
   105      1      0      0000        removed
   110      1      1      0000        removed
   150      0      1      0000        removed
   150      1      1      0000        removed
   150      1      1      0001        removed
```

Figure 7: Example signal trace for the counter model

### 3.2 Digital Control System

The purpose of this example is to demonstrate how the trigger and driver elements can be used to interface an interpreted model of a complex sequential component with an uninterpreted model. In this example, the interpreted model is a microprocessor-based controller with an uninterpreted model of a motor control system. In this model, information effecting the performance of the entire system is transferred back and forth between the two modeling domains.

The uninterpreted elements in this model are a motor controller and a motor. The motor controller periodically asserts the processor's interrupt line. The processor reacts by reading the motor's current speed from a sensor register on the motor controller, calculating the new control information, and writing the control information to the motor controller. Not only does this model provide information about how long it takes the processor to make corrections, but it also gives information about the dynamic response of the system to random variations.

The microcontroller system consists of interpreted models of the 35vee8 microprocessor [8], RAM, memory controller, I/O controller, and clock. The 35vee8 is an eight-bit, RISC-like processor with 64K of addressable memory. The memory controller handles read and write requests issued by the processor to the RAM, while the I/O controller handles read and write request issued by the processor to an I/O device. In the system model, the I/O device is the uninterpreted model of a motor controller. A schematic of the model is shown in Figure 8.

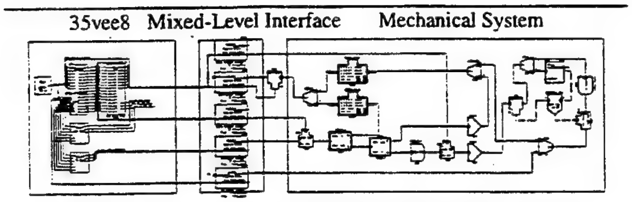


Figure 8: Schematic of control system model

Three triggers and two drivers are used in the mixed-level interface for the control system model. One of the triggers is used to detect when the I/O controller is doing a read or write. The other two triggers are used to collect auxiliary information about the operation, such as the address on the address bus and data on the data bus. One of the drivers is used to create a processor interrupt and the other driver is used to force data onto the data bus when the processor reads from the speed sensor register on the motor controller.

The interrupt driver's program is listed in Figure 9. The program begins by forcing the interrupt line to 'Z' and then waits for a token to arrive. Once a token arrives, the program forces the interrupt line high for ten clock cycles. This condition is accomplished by using a `for-next` statement with a `wait_on_rclk` as the loop body. After ten clock cycles, the program jumps to line 10 where the cycle begins again.

The data driver's program is listed in Figure 10. The program begins by waiting for a token to arrive. If the condition tag field is 1, then "ZZZZZZZZ" is forced onto the data bus. If the condition tag field value is 3, then the value on the `probe_value` tag field of the `in_color_token` input is forced on the data bus. This process is repeated for every token that arrives.

```
driver
10 output Z after 0
20 wait_on_token
30 output 1 after 0
40 for 10
50    wait_on_rclk
60 next
70 goto 10
80 end
```

Figure 9: The interrupt driver's program for the control system

```
driver
10 wait_on_token
20 case_token_is
30     when 1
40         -- sensor not selected
50         output ZZZZZZZZ after 0
60     when 3
70         -- sensor selected for reading
80         dynamic_output_after 0
90 end_case
100 goto 10
110 end
```

Figure 10: The data driver's program for the control system

The I/O trigger's program is listed in Figure 11. This trigger waits until there is a change on the probe. Once there is a change, the program checks to see if the I/O device is being un-selected, written to, or read from. If one of the `when` statements matches the probe value, then its corresponding `output` statement is executed. An output of 1 corresponds to the I/O device not being selected. An output of 2 corresponds to the processor writing control information to the motor controller. An output of 3 corresponds to the processor reading the motor's speed from the sensor register on the motor controller.

Figure 12 shows the results from the mixed-level model as a plot of the sensor output and the processor's control response. Some random error was introduced to the sensor's output to reflect variations in the motor's load as well as sensor noise. The target speed for the system was 63 ticks per sample time. A tick can be thought of as the number of visual markers that have gone by the sensor since the last time its register was read. The system oscillates slightly around this values because of the randomness introduced into the system. At any given time, the sensor can read plus or minus 7 ticks from the actual speed.

This model illustrates how the watch-and-react interface can be used to cosimulate a complex sequential interpreted component such as a microcontroller, and an uninterpreted model. It also illustrates how this cosimulation can be used

```
trigger
10 wait_on_probe
20 case_probe_is
30      when 111
40          -- sensor not selected
50          output 1 after 0
60      when 001
70          -- sensor selected for writing
80          output 2 after 0
90      when 010
100         -- sensor selected for reading
110         output 3 after 0
120 end_case
130 goto 10
```

Figure 11: The I/O trigger's program for the control system

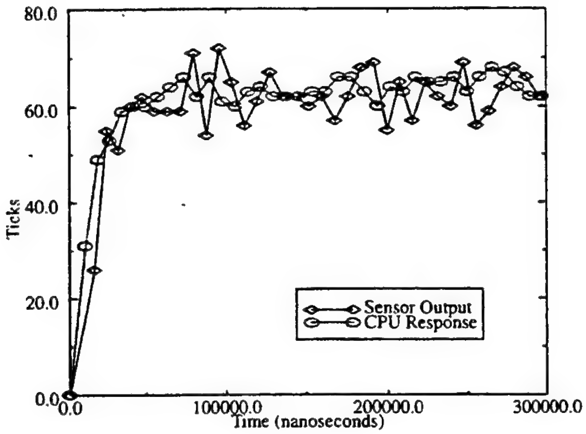


Figure 12: Sensor and processor outputs for the control system

to verify the performance characteristics of a system as its design is refined from initial concept to implementation.

## 4. Conclusions

Using VHDL, it is possible to model systems at many different levels of detail. The levels of detail span the range from high-level performance modeling to low-level behavioral modeling, including an intermediate stage of mixed-level modeling. A graphical design environment called ADEPT (Advanced Design Environment Prototyping Tool) was created to support system evolution from idea to implementation. With this VHDL-based design environment, it is possible to build performance models which can be incrementally refined using behavioral components. Typically in performance models, a mechanism such as token passing is used to represent the flow of information. However, in behavioral models, signals are usually of a less abstract data type, such as bit, std_logic, integer, or real. As a result, an interface between the performance and behavioral modeling domains is necessary for mixed-level modeling. This mixed-level modeling interface resolves the differences in timing and data abstraction between the performance modeling domain (token based) and the behavioral modeling domain (value based).

A specialized version of this mixed-level modeling interface was created in ADEPT specifically for the integration of complex sequential components into performance models. This interface is called the watch-and-react interface. It is based on monitoring a few important signals in a system and then reacting to changes in these signals by generating tokens or forcing signals in the system to appropriate values given the particular situation. Each trigger and driver instance has a program file which specifies how it should operate. Designers can customize these program files to meet their specific mixed-modeling needs. The program files contain scripting instructions which are interpreted by the trigger and driver as the VHDL model simulates. Because the program files are interpreted rather than compiled, they can be changed without having to recompile the VHDL model. This capability provides the ability to taylor the generic trigger and driver elements to any specific application.

## 5. References

[1] Mitra, S. S., *An Interpreted Interface for Hybrid Performance Modeling*, Master's Thesis, Department of Electrical Engineering, University of Virginia, May 1991.

[2] Sahner, R. A., K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems*. Boston: Kluwer Academic Publishers, 1996.

[3] Milne, G. J., and L. Pierre, eds. *Correct Hardware Design and Verification Methods*, Berlin: Springer-Verlag, 1993.

[4] Kumar, S, R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, R. Waxman, "ADEPT: A Unified System Level Modeling Design Environment," *Proceedings of the 1st Annual RASSP Conference*, August 1994, pp. 114 - 123.

[5] Klenke, R. H., M. Meyassed, J. H. Aylor, B. W. Johnson, R. Rao, A. Ghosh, "An Integrated Design Environment for Performance and Dependability Analysis," *Proceedings of the ACM Design Automation Conference*, June 1997 pp. 184-189.

[6] Meyassed, M., *System-Level Design: Hybrid Modeling with Sequential Interpreted Elements*, Ph.D. Dissertation, Department of Electrical Engineering, University of Virginia, January, 1997.

[7] Kant, K. *Introduction to Computer System Performance Evaluation*, New York: McGraw-Hill, 1992.

[8] Williams, R. D. *The 35vee8 Programmers' Reference Manual*, University of Virginia, Version 1.5, 1995.

**Appeared in the Proceedings of the Fall VHDL International Users Forum, 1997, pp. 25-32**

# A Top-down Design Environment for Developing Pipelined Datapaths

Robert McGraw
RAM Laboratories
119 N. El Camino Real, Suite 175
Encinitas,CA 92024
rmcgraw@adnc.com

James H. Aylor
Department of Electrical Engineering
University of Virginia
Charlottesville, VA 22903
jha@virginia.edu

Robert H. Klenke
Department of Electrical Engineering
University of Virginia
Charlottesville, VA 22903
rhk2j@virginia.edu

## 1. Abstract

*This paper presents a design environment for cycle-based systems, such as microprocessors, that permits modeling of these systems at various levels, from the abstract system level, through the detailed RTL level, to an actual implementation. The environment allows the models to be refined to lower levels in a step-wise manner. The environment provides the ability to obtain meaningful metrics from abstract models of a processor's architecture. This capability allows design alternatives to be evaluated earlier in the design cycle, thus eliminating costly redesign and reducing the processor time to market.*

## 2. Introduction

Currently within the design community there is an increasing interest in the development of methodologies which reduce the time to market for a given system under development. One area of particular concern deals with the development of application specific processors [1]. With integrated circuits projected to reach the size of over 100 million transistors per die by the turn of the century [2], this increasing complexity must be handled properly so as not to adversely affect processor design time. One way to address this problem of complexity management is through the use of a top-down design methodology.

Top-down design methodologies have been used to design digital hardware design since the early 1970's [3]. A top-down design methodology follows a design from the top level, usually the specification level, of detail down to a detailed implementation. Model refinement in these methodologies works by having each level of detail serve as the design specification for the level of detail immediately below. It is acknowledged that if this hierarchical chain can be verified from one level of detail to the next, the resulting behavioral implementation will be "*right the first time*" [4]. Being able to develop systems that work on the first pass in a timely manner helps address the time to market problem. Unfortunately, there exists a lack of modeling environments which promote complete top-down design and refinement of processors from the system level.

This paper presents a timed cycle-based design environment which is geared toward the development of pipelined datapaths for processors and other synchronous systems. This cycle-based environment permits the processor designer to model and hierarchically refine pipelined processor datapaths from the system level down through the RTL level until a behavioral implementation has been developed. This paper focuses on the modeling and development of pipelined datapaths because most modern processor architectures contain considerable pipelining. The remainder of this paper is organized as follows; Section 2 presents a background of existing processor design environments. Section 3 presents an overview of the new design environment proposed herein. Section 4 describes the intermediate level modeling capability of the environment that provides a link between the abstract system level of modeling and the detailed functional level model. Finally, Section 5 presents an example of modeling a MIPS R4000 processor using the environment and Section 6 presents some conclusions.

## 3. Existing environments and methods

For a processor design environment to completely support top-down design and refinement, the environment must have some means of developing a system level processor model, some means of refining the system level model to the RTL level, and some means of providing abstract control to the datapath in order to obtain meaningful results from the model. At each level of design detail, different architectural analyses can be performed as detailed in Figure 1. For instance, at the system level, datapath control is often provided through the use of random distributions to exercise all model paths. Resulting analyses which can be performed include determination of cycle time and critical paths. At the RTL level, the design is very detailed and control is provided by a explicit control unit. At the RTL level, all functional and detailed performance metrics can be obtained. The need for a methodology and environment which supports the modeling and refinement of both system level and RTL level datapath models has been expressed in the literature [5,6].

Existing commercial design methodologies use a variety of tools to analyze designs at varying levels of detail. For example, Sun Microsystems uses architecture-specific simulators such as the UltraSPARC Performance Simulator (UPS) [7] to examine architectural trade-offs at a functional level. The UPS is a trace-driven simulator designed to simulate the Ultra-SPARC microarchitecture at a functional, RTL level of modeling. IBM uses several modeling tools to satisfy different parts of its design methodology during the development of its PowerPC line of processors. IBM examines architectural trade-offs at the functional level of detail by using the Basic RISC Architecture Timer (BRAT) [8]. The BRAT tool is an architecture-specific simulator. IBM also developed processor models using Verilog and their propriety Design Structure Language (DSL) which were used to analyze architectural trade-offs at the both the system and functional levels.The DEC design methodology for the 100 MHz CISC NVAX processor and the 200 MHz RISC Alpha AXP 21064 processor [9,10] included the analysis of the processor architecture starting at the RTL level using Digital's in-house hardware description

To Appear in the Proceedings of the Design Automation Conference, 1998

**Design Flow**

| Modeling Level | Datapath Control | Model Input | Type of Analyses |
|---|---|---|---|
| System Level | Distribution-based | Distribution-based | Cycle, setup and hold times, Critical Path Analysis |
| Intermediate System/RTL | Instruction-based Control provided through reservation tables or RTL level descriptions | Instruction Trace or Instruction Mix Driven | Cycle-time, Critical Path Analyses, Latency, Throughput, Concurrency, Register Setup and Hold Times, Determination of MIPS bounds |
| Register Transfer Level (functional) | Based on Modeled Control Unit or Datapath Information | Instruction Trace or Instruction Mix Driven | All Performance and Functional Analyses |

**Figure 1. Design flow for methodologies**

language (DECSIM) based simulator. AMD's Am29000 processor was modeled at the functional level, using a specifically designed, C-based architectural language, and at the gate level to guide logic design [11]. Additional processor design methodologies currently being used deal with design starting at the functional level [12,13]. These methodologies are similar to bottom-up design strategies in that they are often based on existing architectures.

Using existing methods, the complete design of a processor cannot be performed in the same modeling and simulation environment. The current methodologies require the construction of multiple, disjoint, processor models at the system level and RTL level [14]. Typically these methodologies create system level models on a processor-by-processor basis using some type of modeling language or hardware description language. When a more detailed model of the processor is required, a new model is developed at the RTL level. There are several reasons for the creation of multiple models. First, processor modeling tools above the RTL level are relatively non-existent. Second, along with the need to address design refinement issues, existing environments do not have suitable methods for controlling an abstract datapath to produce meaningful results.

Several approaches have attempted to address processor and datapath modeling above the RTL level. Zhang and Grunbacher [5] have developed a Petri Net based design approach for pipelined processors. This approach allows for a design to be modeled at the system level through the use of Petri Nets. In addition, Razouk [6] has developed a timed Petri Net approach for detailed modeling of a processor design at the system level. Unfortunately, these methods lack the ability to link system level modeling environments to the RTL level of development.

This paper presents a timed-cycle-based design environment which allows for the modeling of processor datapaths above the RTL level. In particular, this methodology and environment provides datapath development constructs and control methods which link system level and RTL level models together through the use of an intermediate system/RTL level modeling domain. This intermediate system/RTL level domain, detailed in the shaded row of Figure 1, consists of a model of execution and datapath control methods which allow for the analysis of pipelined datapaths.

## 4. Environment and Model of Execution

A timed cycle-based processor design environment which specifically addresses the development of pipelined datapaths has been constructed. This environment supports system level processor modeling using abstract datapath constructs and mechanisms to control the datapaths. This environment addresses model refinement issues by providing modeling constructs and abstract control methods which bridge the system/RTL level modeling gap.

This design environment is based on the ADEPT performance modeling environment [15]. ADEPT is based on the VHSIC Hardware Description Language (VHDL) and provides a modeling environment where high-level models can be refined down to an implementation in an integrated manner. In the ADEPT environment, a system model is constructed by interconnecting a collection of *ADEPT modules*. The modules model the information flow, both data and control, through a system. Each ADEPT module is implemented in VHDL and communicates with other modules by exchanging *tokens* which represent data being transmitted in the system. The ADEPT modeling modules communicate via a four-state token passing protocol (present, acknowledged, released, removed). This protocol provides fully interlocked handshaking between elements. This type of asynchronous handshaking protocol is needed because the communications between the existing ADEPT modules is inherently asynchronous in nature. The VHDL code generated by ADEPT can be simulated using any IEEE 1076-87 compliant VHDL simulator. Facilities and programs to collect and analyze the simulation results are provided as part of the ADEPT system.

### 4.1 Design Flow and Datapath Control

The timed cycle-based environment augments the existing ADEPT environment to allow for the modeling of cycle-based systems, while still including the concept of asynchronous delay for combinational elements. The design flow using this timed cycle-based environment takes an instruction set architecture and refines it using modeling constructs of increasing detail through the RTL level down to a behavioral implementation. The cycle-based modeling constructs support datapath modeling at, or above, the RTL level. In addition, the existing capabilities for mixed-level modeling in ADEPT [15] allow RTL level models to be refined to an actual implementation in a step-wise manner. The modeling levels which are supported by the cycle-based modeling constructs include an abstract system level, an intermediate system/RTL level, and an RTL level. These modeling levels are unique in that they are exercised using different means of datapath control as more detail is entered into the datapath model. Figure 1 denotes the modeling levels along with methods for controlling those levels, means of exercising models developed at those levels, and the types of functional and performance analyses which can be performed at each level.

The system level modeling domain supports a very high-level of abstraction (almost all data and control have been abstracted away). This particular level of design can be equated to a "block-diagram" level of design detail. At this particular level, all of the clocked elements (registers, memories) are required to be present in the design. These elements receive or source the information tokens on every cycle. In addition, the combinational elements, between the clocked elements, are modeled simply as delay elements. The system level modeling constructs currently included in the environment include clocked register constructs and various routing elements which mainly deal with value-less (uninterpreted) tokens. An example of a system level model of a four-stage pipeline is shown in Figure 2(a). Datapath routing at this level is accomplished by using various stochastic methods. At this level, datapath control is provided by using stochastic distributions to make routing decisions as tokens arrive *at the routing elements.* The main goal of modeling at such a level is to ensure that the information flow between clocked elements meets the cycle time requirements.

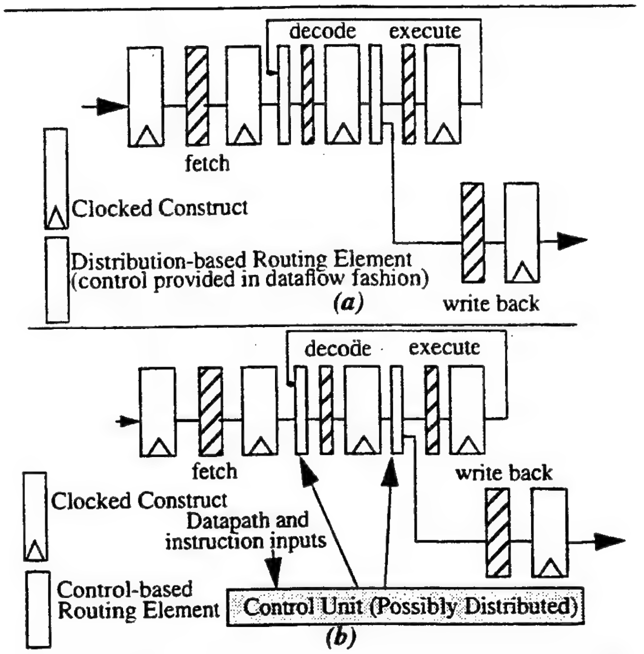

decode
execute
fetch
Clocked Construct
Distribution-based Routing Element (control provided in dataflow fashion)
(a)
write back
decode
execute
fetch
write back
Clocked Construct
Datapath and instruction inputs
Control-based Routing Element
Control Unit (Possibly Distributed)
(b)

**Figure 2. 4-stage pipeline (a) system level, (b) intermediate**

The RTL level of modeling is much more detailed than the system modeling level. The RTL level modeling constructs include various clocked memory and register elements along with processor routing elements. An example of a RTL level model is shown in Figure 2(b). These RTL level modeling constructs model existing hardware elements such as multiplexers, demultiplexers and combinational logic using a one to one mapping of hardware signals to tokens or token values. The RTL level constructs are value-based. The constructs at the RTL modeling level route tokens based primarily on token values. The control for the RTL level datapaths is typically provided through some modeled control unit. In addition to having the responsibility of routing tokens from register to register, the RTL modeling level unclocked constructs also have the capability to operate on data (found on the token color fields).

The intermediate system/RTL level modeling constructs are the key to the environment in that they provide a link, through refinement, between the system level of modeling and RTL level of modeling. This intermediate system/RTL level modeling constructs and control methods are discussed in Section 4. By providing constructs which gradually incorporate more detail, the cycle-based design environment facilitates step-wise refinement from the system level to the RTL level.

## 4.2 Models of Execution

In order to communicate between various cycle-based modeling constructs, each construct must have a consistent model of execution. The model of execution refers to the way in which the modeling constructs of this environment communicate with each other. Because the modeling constructs must actually represent real systems or elements in a synchronous environment, models of execution for two types of elements are needed: clocked constructs (for synchronous elements) and unclocked constructs (for combinational elements).

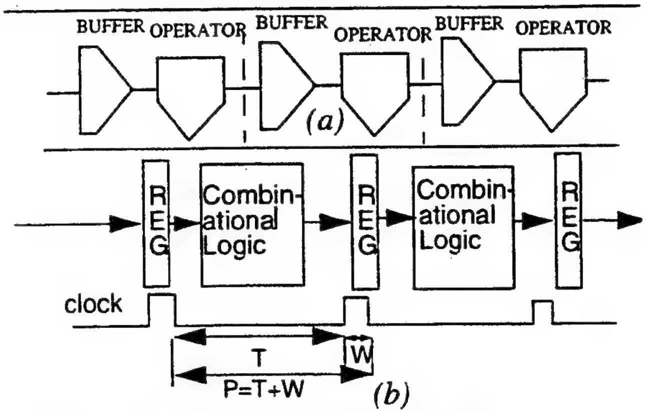


**Figure 3. Dataflow representation of pipeline**

Typically, existing processor datapaths can be represented using pipelined stages in a manner similar to Figure 3 [16]. Such a pipelined architecture is often implemented through stages of clocked elements (registers) followed by unclocked elements (combinational elements) as shown in Figure 3(b). Existing cycle-based environments typically map this pipeline architecture into the representation of Figure 3(a). Figure 3(a) shows each pipeline stage as being comprised of buffer elements followed by some type of operator element. The concept of buffering of information between modules is important because it is this buffering which separates the pipeline stages. In terms of the ADEPT four-way handshaking protocol, the buffer is the element that acknowledges the receipt of the token at the next stage of the pipeline. The operator element is viewed as an element which simply "operates" on arriving information before passing it on to subsequent pipeline stages. The operator modules do not buffer or acknowledge the receipt of information. The operator elements have an asynchronous delay representing combinational blocks and are known as the unclocked elements. In addition, the buffer elements are only allowed to acknowledge receipt of information on cycle boundaries. These are known as the clocked elements.

The model of execution for the unclocked elements is fairly straightforward. The unclocked constructs operate via the four-way interlocking handshake for asynchronous elements. These constructs map their inputs to their outputs using some type of control mechanism. This control mechanism may require inputs to be joined, synchronized, or forked in order to map them to the outputs. These constructs are also unbuffered in that they do not generate an acknowledgment upon the receipt of information. These constructs simply operate on arriving information and pass the information to the next construct.

The model of execution for clocked constructs is more complicated. The clocked elements are synchronized by some clock signal (to identify the cycle boundaries), yet these constructs must maintain a four-way interlocking handshake so they can communicate with the unclocked elements. In addition, these elements must contain buffering in order to acknowledge receipt of information at the cycle boundaries for each pipeline stage. For this reason, the model of execution for the clocked elements handles the four-way handshake, the buffering and acknowledgment of information, and the synchronizing of the inputs and outputs with respect to some type of clock signal.

The model of execution for both the unclocked and clocked

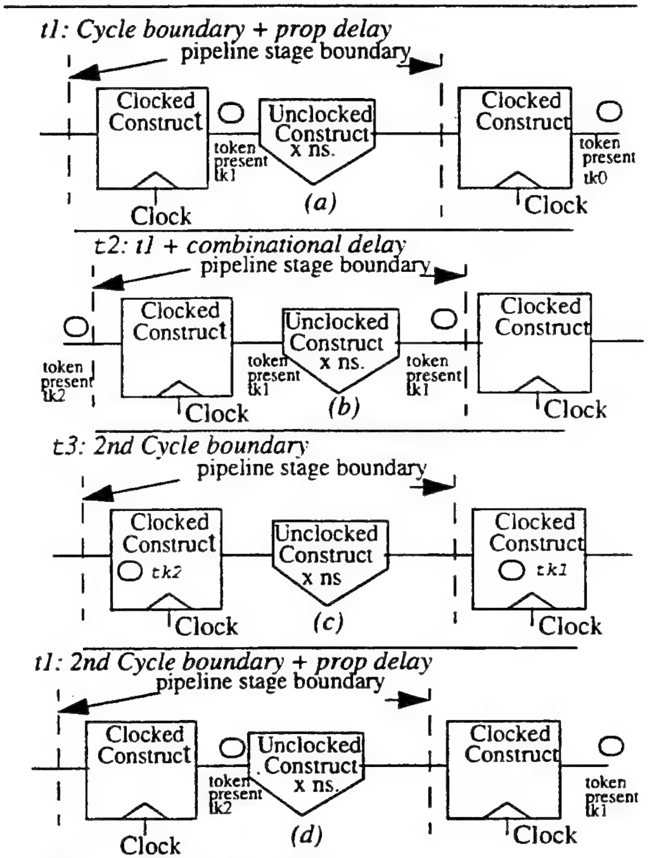


**Figure 4. Model of execution**

constructs for the cycle-based design environment is demonstrated in Figure 4 using a single pipeline stage. Figure 4(a) shows the clocked constructs outputting a token (*tk0,tk1*) at the cycle boundary after the propagation delay of the clocked constructs. This is represented by the token being present at the outputs of the clocked constructs. Figure 4(b) shows the token (*tk1*) propagating through the unclocked constructs after a delay of *X* (equal to the unclocked combinational delay). This results in the token, *tk1*, being present at the output of the unclocked construct. Because the unclocked constructs are unbuffered elements and do not generate their own acknowledgment, the input of the unclocked constructs still has token, *tk1*, present. The tokens remain in these "present" states until the cycle boundary is reached.

At the cycle boundary (determined when the clock signal is enabled), the clocked constructs copy the values on their input tokens to an internal token and finish the four-state handshake on their inputs. The clocked constructs then place their internal tokens on their outputs after accounting for the propagation delay *only* if those outputs are clear. This is the normal operating model of execution for the clocked constructs.

The models of execution for the clocked and unclocked constructs were verified using several basic architecture configurations. These basic configurations included linear pipelines, linear pipelines with single feedback loops, and linear pipelines with multiple feedback loops. In addition, each model of execution's ability to handle a stalled pipeline (due to resource contention issues or multiple cycle delay stages) has also been examined and verified.

## 5. Intermediate System/RTL Level Modeling

This new environment is set apart from the existing environments in that it provides an intermediate system/RTL level of modeling constructs which bridges the system level to RTL level modeling gap for abstract processor datapaths.

### 5.1 Intermediate System/RTL Level Modeling Constructs

Datapaths developed using the intermediate system/RTL level modeling constructs can provide the designer with a more detailed datapath analysis than can be found using only a system level model. While continuing to allow the designer to perform cycle-time and critical path analyses, datapaths which are developed using the more detailed intermediate system/RTL level modeling constructs also allow the designer to examine concurrency issues and perform latency and throughput analyses. Also, the system/RTL modeling level can permit the designer to obtain an estimated value for *instructions per second* before a detailed design or a complete compiler for the processor are developed.

The intermediate system/RTL modeling level constructs route the datapath information based on the desired datapath routes needed to satisfy a particular instruction. Typically these datapaths will be exercised using a statistical instruction mix, although an instruction trace can also be used. Each element of the system/RTL level datapath receives the active instruction, or instructions, for the current cycle. Because a modeled control unit is typically absent at early stages of the design cycle, the datapath control must be solely based on this instruction and its associated instruction fields. The current instruction is provided through the use of a colored information token. The control for the system/RTL level datapaths is dependent upon this current instruction and provided in two ways, depending upon the type of datapath and analyses required by the designer. Control for un-pipelined datapaths is provided based on the register transfer description for each instruction. Control for pipelined datapaths is provided using the reservation tables which describe the stage to stage information flow for each instruction. The reservation table-based control methods and modeling constructs are described in Section 5.2.

### 5.2 Reservation Table Control Methods

The goals of analyzing such a pipelined datapath would be to obtain latency and throughput information as well as a bounds for *instructions per second* for a pipelined execution unit under a given instruction mix or workload. One way of providing the control information for pipelined units is to make use of design methods concerning the design of pipelined execution units. In order to analyze the operation of pipelined execution units (such as integer pipelines and floating point units) system designers often use reservation tables [17,18]. Reservation tables are used to specify the use of given resources used by a instruction as it proceeds through the pipeline. Reservation tables can be used to determine instruction latency, or how long an instruction has to wait at the "head" of the pipeline before entering without causing resource contentions. These reservation tables can be used to give the designer a rough idea of attainable throughput and latency metrics concerning any pipelined unit.

The system/RTL level modeling constructs allow the designer to encode these reservation tables in a file. Figure 5 shows the reservation table for an integer instruction for the four-stage DLX pipeline of Figure 6[19]. The intermediate system/RTL level model of the four-stage pipeline is shown in Figure 6. The coded reservation tables are accessed by the intermediate system/RTL level routing elements and used to control the pipelined datapaths on a cycle-by-cycle basis.

The reservation tables are employed at the pipeline's clocked constructs to control instruction initiation within the pipeline. The pipehead_cyc element, shown in Figure 7, is the clocked element which has been developed to govern instruction initiations. The pipehead_cyc modeling construct requires four generic properties: *i_tag*, *trig_tag*, *delay*, and *filename1*. The *i_tag* property specifies the token color tag on which the instruction information is contained. The *delay* property specifies the propagation delay tokens encounter while passing through the pipehead_cyc element. The *filename1* property specifies the file which contains the names of the coded reservation table data files and reference numbers for all pipeline reservation tables which are used in the pipeline model.

The pipehead_cyc element is placed at the head of the top-level pipeline. When instructions arrive at the pipehead_cyc construct, they are checked for resource conflicts with all resources in the pipeline. First, the pipeline status reservation table is accessed. This pipeline status reservation table contains the status information (stage and cycle markings) for the pipeline referenced by the pipehead_cyc construct. This pipeline status reservation table is intersected with the reservation table of the incoming instruction to determine if a resource contention will occur if that instruction is initiated. If a resource contention will occur if the instruction is initiated, then no initiation is made for that cycle and the instruction is left on the pipehead_cyc element's input. This allows the same instruction to be presented on the subsequent cycle.

The reservation tables are also utilized at the pipeline's unclocked routing elements to control the stage-to-stage routing for each instruction within the pipeline. The unclocked routing units have their outputs bound (using defined properties and net interconnections) to different stages of the modeled pipeline. Tokens arriving at the unclocked elements are routed by accessing their reservation tables, and identifying the stage(s) to which they should be routed for that cycle. An example of such a routing element is the piperoute2 element, which is shown in Figure 8. The piperoute2 is used to route tokens internal to the pipeline execution units using reservation tables. This element requires several generics in order use reservation tables to assist in routing tokens. The *i_tag* and *filename1* properties are the same as those found in the pipehead_cyc element. In addition to these properties, the piperoute2 element also has output binding properties *Outbindings1* and *Outbindings2*, a *max_inst* property, a *maxclkcyc* property, a *pipelength* property and a *cyc_no_tag* property. The *max_inst* property specifies the maximum number of instructions to be handled by the pipeline containing this particular element. The *maxclkcyc* property specifies the maximum number of cycles required to complete any instruction. The *pipelength* property specifies the number of stages in the pipeline. The *cyc_no_tag* property specifies the token color tag field which contains the cycle count for each instruction. The *Outbindings* properties specify the stage connectivity for each output. These properties are arrays which list the stages which connect to the current stage. For example, the four-stage DLX pipeline of Figure 6 contains a piperoute2 construct after its decode stage (stage 2). This routing element is required because information needs to be routed to the write back (stage 4) or execution (stage 3) stages after the decode stage. For this reason, the *Outbindings* properties of the piperoute2 construct are assigned to stages 3 and 4 respectively. When the token arrives at the piperoute2 element, its reservation table is accessed and the token is routed to the output which is referenced in the reservation table for that cycle.

| cycle<br>stage | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Fetch | X | | | | |
| Decode | | X | | X | |
| Ex | | | X | | |
| WB | | | | | X |

Coded Reservation Table

i1
10000
01010
00100
00001

Figure 5. Reservation Table and Coded File for Figure 6

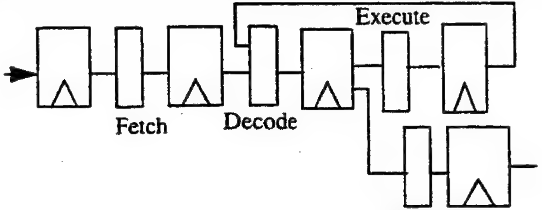


Figure 6. Block Diagram of Pipelined Datapath

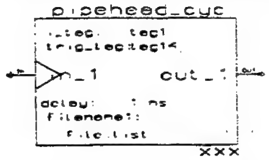
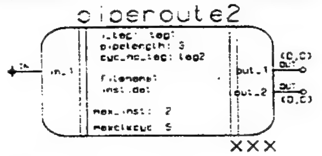


Figure 7. Pipehead_cyc Modeling Construct

## 5.3 Hierarchical Modeling Using Reservation Tables

In order to facilitate top-down design and refinement, the timed cycle-based environment has the capability of modeling hierarchical pipelines using the reservation table control methods. The control methods developed allow for the insertion of a "low-level" pipeline into a top-level pipelined datapath.

The pipelined cycle-based constructs were used to model a five-stage DLX pipeline with multiple execution units. Figure 9 shows the five-stage DLX pipeline (fetch, decode, execute, memory access, write-back) where execution unit 2 (Ex-2) represents a multi-function floating point unit that has its own underlying reservation table. Ideally, once this pipelined stage has been designed, it would be desired to hierarchically replace the single Ex-2 stage in the top-level pipeline with the 3-stage multifunction pipeline. In addition, it is also necessary to perform all routing at this "lower-level" by routing this 3-stage pipeline locally using its own reservation table. The original top-level reservation table is then altered to reflect the extra cycles spent in the multi-function



(3,0) Property Assignment
(4,0) for 4-stage example

Figure 8. Piperoute2

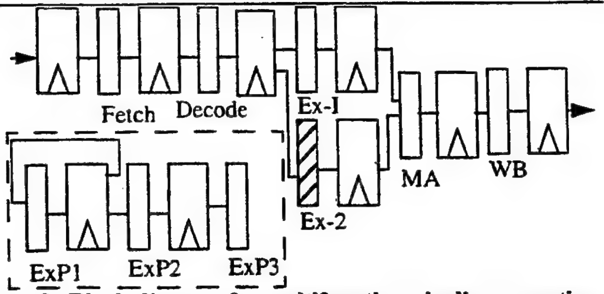


Figure 9. Block diagram for multifunction pipeline operation

Ex-2 unit.

The pipehead_cyc modeling constructs have the capability to examine all levels of pipeline hierarchy to decide when instruction initiations can be performed at the head of the top-level pipeline. By allowing the pipehead_cyc modeling construct at the head of the top-level pipeline to examine the reservation tables for all levels of pipeline hierarchy, refined lower-level pipelines can be simply "plugged in" to the top-level datapath model.

## 6. Example --- MIPS R4000 Processor

The timed cycle-based modeling environment's intermediate system/RTL modeling level has been verified through the use of several examples. One such example involves the modeling of a MIPS R-4000 processor [20]. The MIPS R-4000 is a 9-stage pipelined processor. These stages include: instruction fetch 1, instruction fetch 2, register fetch, execution, data fetch 1, data fetch 2, tag check and, write back. The execution units used for the execution pipeline stage include both an single stage integer execution unit and a floating point unit consisting of 8 pipelined stages. The MIPS R-4000 was modeled in hierarchical fashion with separate reservation tables developed for the integer and floating point execution units.

The reservation tables for both the top-level MIPS DLX pipeline and the internal pipeline execution unit were developed by hand for each instruction. During simulation, each modeling element accesses these reservation tables to determine if an instruction can be initiated and where information needs to be routed to. The MIPS R-4000 model was exercised using the SPEC95 benchmarks [21]. The benchmark instruction traces were obtained by compiling the SPEC95 source code on a Silicon Graphics MIPS R4000 machine and outputting the symbolic assembly instruction trace. This assembly language instruction trace was then mapped to instruction tokens entering the DLX pipeline. The MIPS R-4000 DLX model was simulated using a Mentor Graphics QuickVHDL simulator on a Sun Sparc-10 workstation. The simulation showed that the R-4000 processor executing the SPEC benchmark tomcatv.f, had a millions of instructions per second (MIPS) value of 75.76 using a 100 MHz clock. The published performance rating for the 100 MHz SGI R4000 was 76.5 indicating the abstract model was useful in obtaining a ballpark performance metric for millions of instructions per second. The model simulates at 16.4 cycles per minute of CPU time. It should be noted that this model did not take into account such issues as cache hits and misses and interrupts. Because this model was constructed at the intermediate system/RTL modeling level, statistical probabilities were used to help predict instruction branching. Exact branching values were not used because this model is an abstract model and does not contain the detail required to obtain those values. By using distributions to predict when branches could by taken, the model was able to use the SPEC benchmark traces in sequence to obtain a representative workload.

## 7. Summary and Conclusions

This paper presented a timed cycle-based design environment which provides a means for modeling and simulating processor datapaths at high levels of design abstraction. This environment was made possible by developing modeling constructs and abstract control methods which facilitate the modeling and control of processor datapaths above the RTL level. The methods for controlling the abstract processor datapath models are rooted in existing processor design methods and have been extended to assist in exercising meaningful processor models at early stages of the design. By obtaining meaningful metrics from abstract models of the processor's architecture, design decisions can be evaluated earlier in the design cycle, thus eliminating costly redesign and reducing the processor time to market.

## 8. References

[1] Proceedings, First Annual RASSP Conference, August 1994.

[2] Heaton, J., "Simulation - A Key to Smart Design,". Proceedings, Institution of Electrical Engineers, 1995.

[3] Rose, Charles. "The What and How of Top-Down System Design" TD Technologies, 1993.

[4] Transcend Promotional Document, TD Technologies, Inc.

[5] Zhang, Q.and H. Grunbacher. "Petri Nets Modeling in Pipelined Microprocessor Design," Applications of Theory of Petri Nets, pp. 582-591. 1993.

[6] Razouk, Rami R. "The Use of Petri Nets for Modeling Pipelined Processors," 25th ACM/IEEE Design Automation Conference. pp. 548-553.

[7] Tremblay, Maturana, Inoue, and Kohn. "A Fast and Flexible Simulator for Micro-architecture trade-off analysis on Ultra-Sparc-I," 32nd Design Automation Conference, 1995. pp. 2-6.

[8] Poursepanj, et al."The PowerPC 603 Microprocessor: Performance Analysis and Design Trade-offs," IEEE Spring COMPCON 1994 pp. 316-323.

[9] Dutton, Todd A. "The Design of the DEC 3000 Model 500 AXP Workstation," IEEE Digest of Papers, COMPCON, Spring 1993. pp 449-454.

[10] Peng, Donchin, Yen. "Design Methodology and CAD Tools for the NVAX Microprocessor," IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1992 pp. 310-313.

[11] "Simulation in the design of the Am29000 microprocessor," Electronic Engineering. November 1987. pp. 44-52.

[12] Taylor, Rekow, Radke, and Thompson. "A 100 MHz Floating Point/ Integer Processor," IEEE 1990 Custom Integrated Circuits Conference. pp. 24.5.1-24.5.4.

[13] Narita, Arakawa, Uchiyama, and Kawasaki. "Design Methodology for GMicro/500 TRON Microprocessor," IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1993 pp. 253-257.

[14] Franke, D., Purvis, M., "Hardware/Software Codesign: A Perspective", 13th International Conference on Software Engineering, May 1991, pp. 344-352.

[15] Klenke, R. H., M. Meyassed, J. H. Aylor, B. W. Johnson, R. Rao, A. Ghosh, "An Integrated Design Environment for Performance and Dependability Analysis," Proceedings of the ACM Design Automation Conference, June 1997 pp. 184-189.

[16] Kogge, Peter M. The Architecture of Pipelined Computers. Hemisphere Publishing Corporation, 1981.

[17] Stone, Harold S. High-Performance Computer Architecture. Addison-Wesley Publishing. 1993.

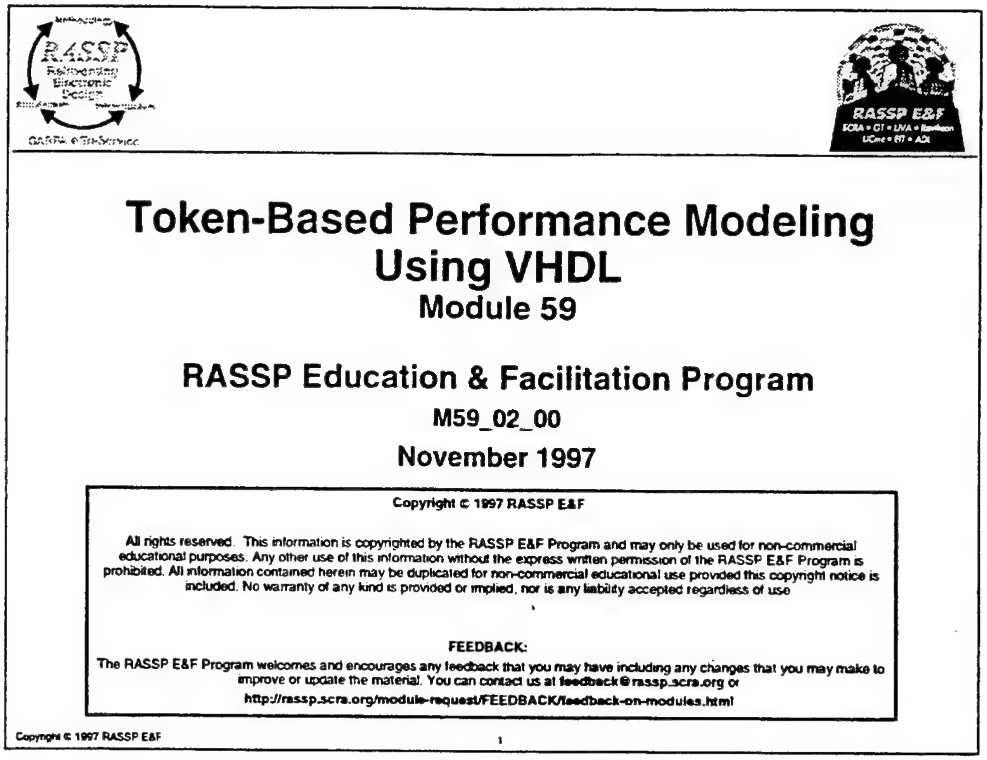
[18] Hayes, John P. Digital System Design and Microprocessors. McGraw-Hill, Inc. 1984.

[19] Hennessey, John L. and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, Publishers, San Francisco, Ca. 1996.

[20] MIPS R-4000 User's Manual, Silicon Graphics, Inc.

[21] Standard Performance Evaluation Corporation. 1995 Benchmarks.

# APPENDIX B –Relevant RASSP E&F Educational Module



# Token-Based Performance Modeling Using VHDL

## Module 59

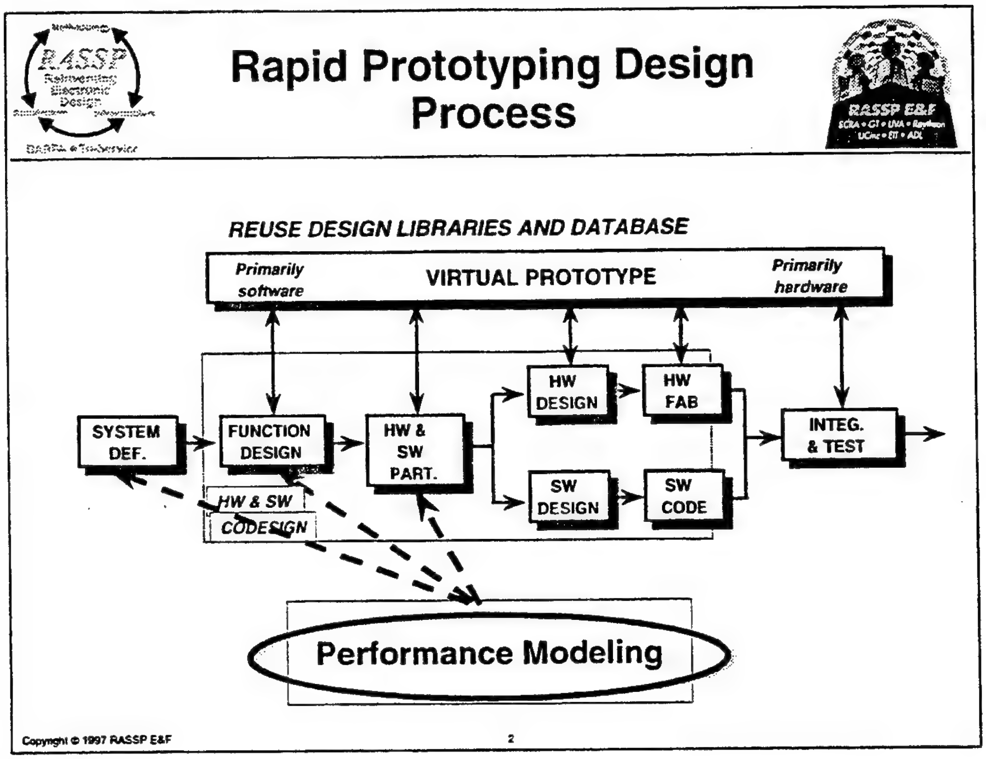
## RASSP Education & Facilitation Program

M59_02_00

November 1997

**FEEDBACK:**

The RASSP E&F Program welcomes and encourages any feedback that you may have including any changes that you may make to improve or update the material. You can contact us at **feedback@rassp.scra.org** or **http://rassp.scra.org/module-request/FEEDBACK/feedback-on-modules.html**



## Rapid Prototyping Design Process

*REUSE DESIGN LIBRARIES AND DATABASE*

*Primarily software*

VIRTUAL PROTOTYPE

*Primarily hardware*

SYSTEM DEF.

FUNCTION DESIGN

*HW & SW CODESIGN*

HW & SW PART.

HW DESIGN

HW FAB

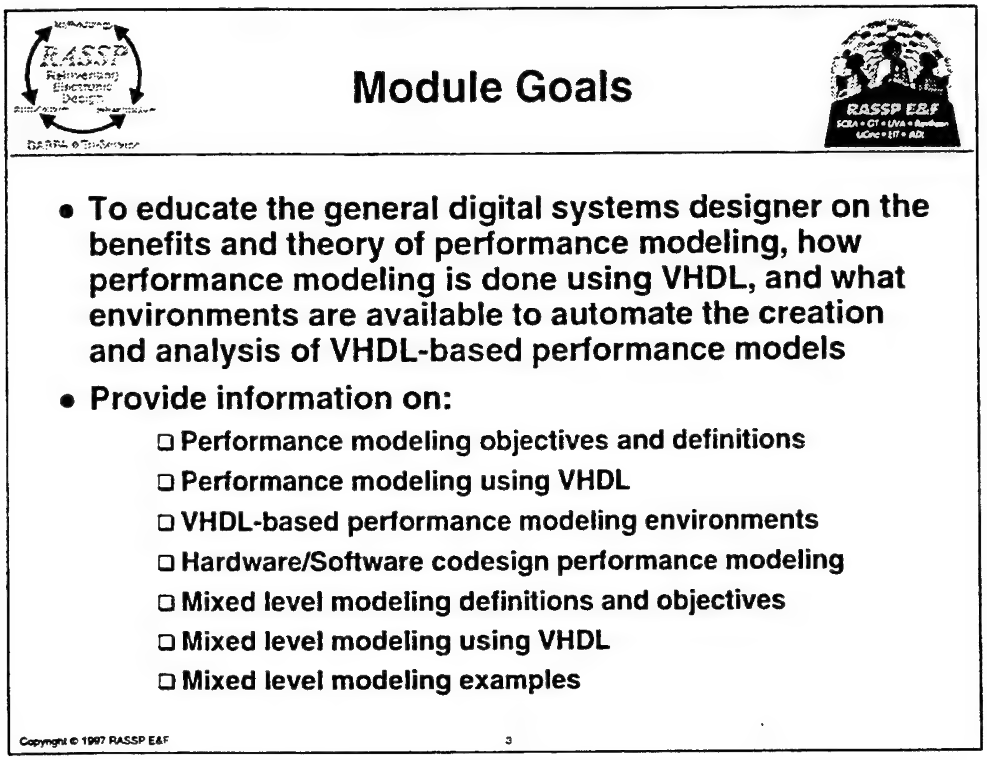
SW DESIGN

SW CODE

INTEG. & TEST

Performance Modeling

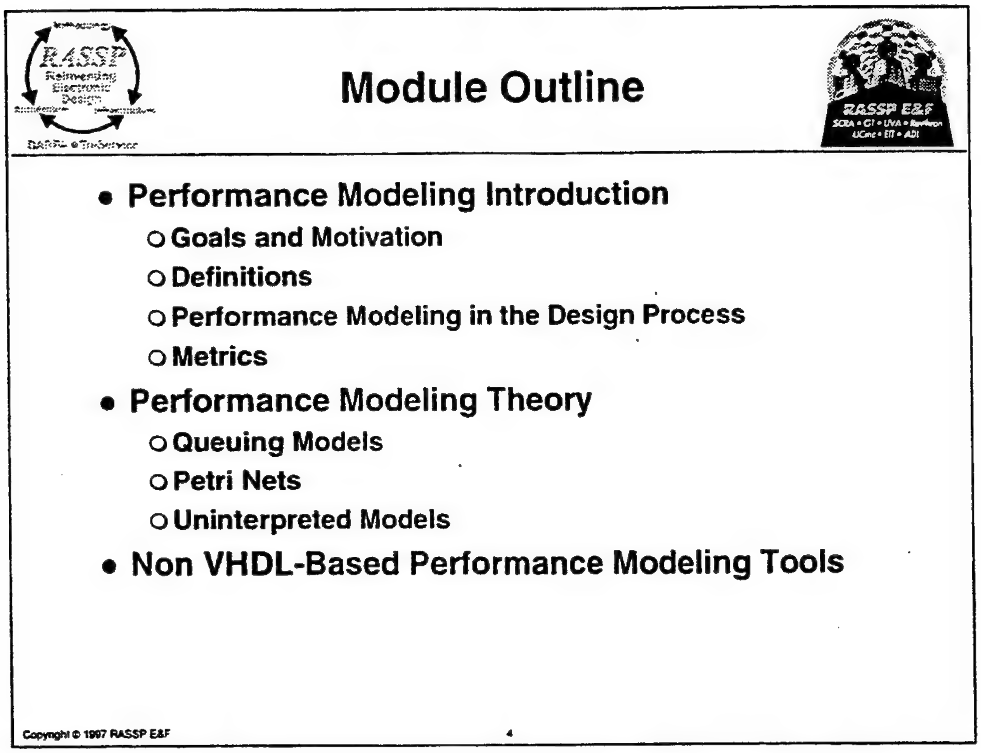
This slide shows the application area for performance modeling. It will be explained in more detail later in the module.



## Module Goals

- To educate the general digital systems designer on the benefits and theory of performance modeling, how performance modeling is done using VHDL, and what environments are available to automate the creation and analysis of VHDL-based performance models
- Provide information on:
    - ☐ Performance modeling objectives and definitions
    - ☐ Performance modeling using VHDL
    - ☐ VHDL-based performance modeling environments
    - ☐ Hardware/Software codesign performance modeling
    - ☐ Mixed level modeling definitions and objectives
    - ☐ Mixed level modeling using VHDL
    - ☐ Mixed level modeling examples

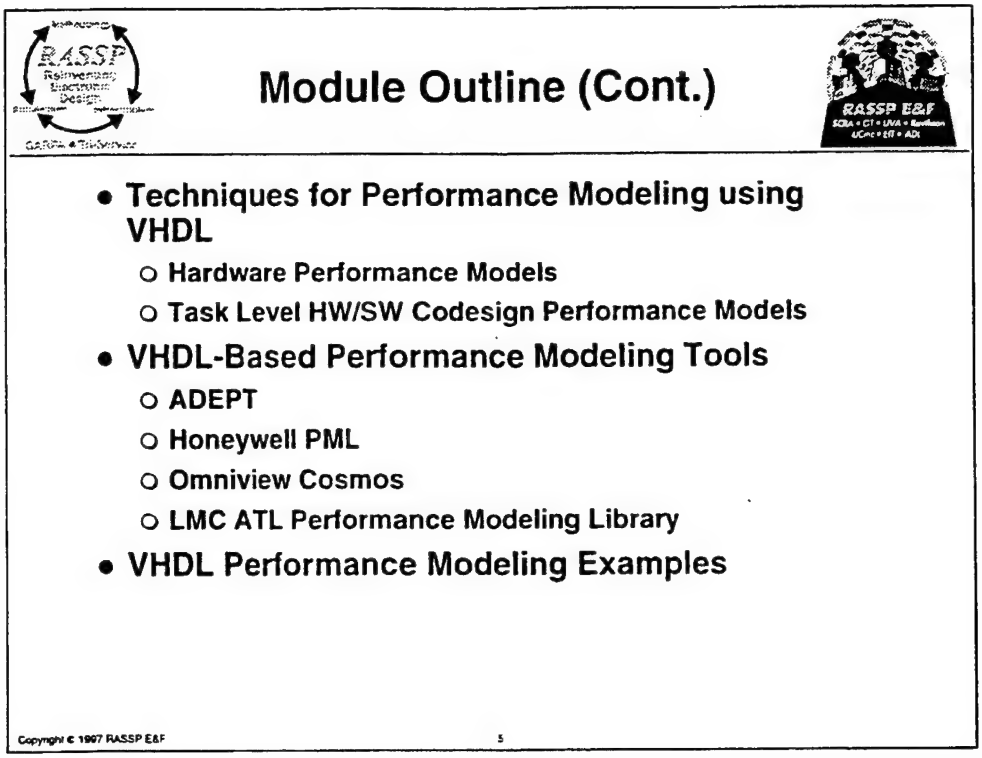
No notes necessary.



# Module Outline

- Performance Modeling Introduction
  - Goals and Motivation
  - Definitions
  - Performance Modeling in the Design Process
  - Metrics
- Performance Modeling Theory
  - Queuing Models
  - Petri Nets
  - Uninterpreted Models
- Non VHDL-Based Performance Modeling Tools

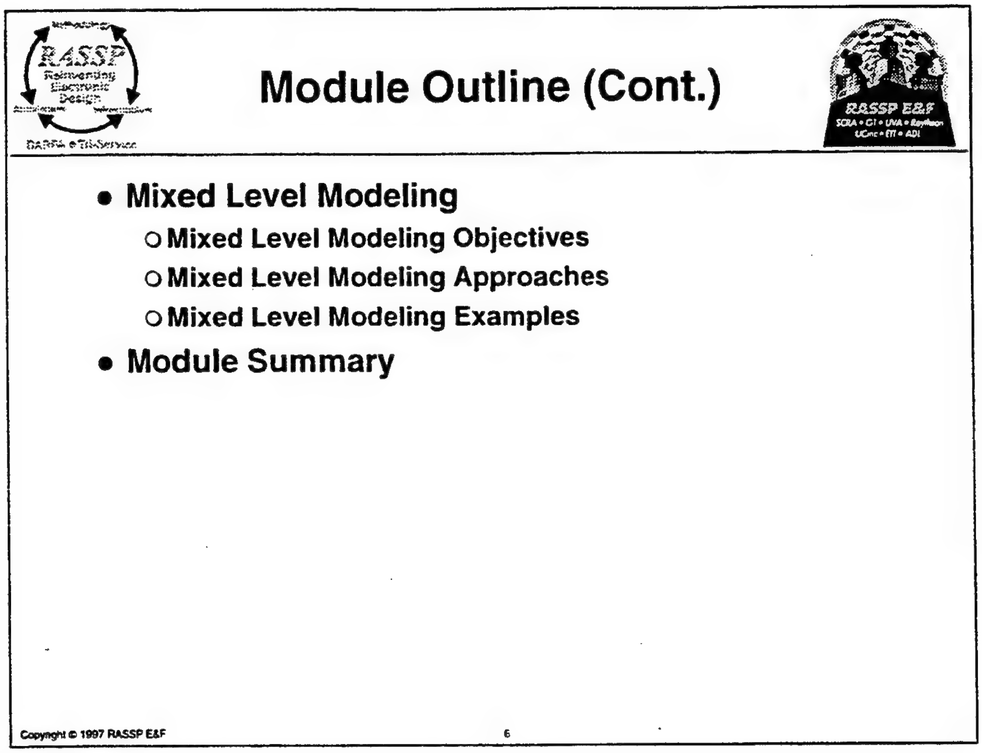
No notes necessary.



## Module Outline (Cont.)

- Techniques for Performance Modeling using VHDL
  - Hardware Performance Models
  - Task Level HW/SW Codesign Performance Models
- VHDL-Based Performance Modeling Tools
  - ADEPT
  - Honeywell PML
  - Omniview Cosmos
  - LMC ATL Performance Modeling Library
- VHDL Performance Modeling Examples

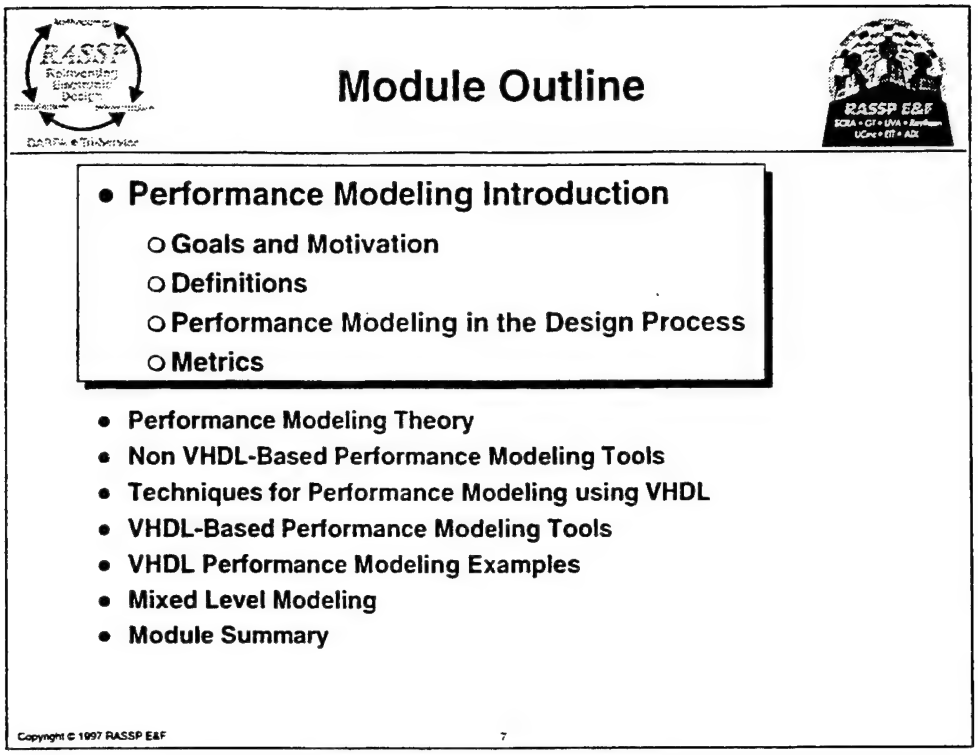
No notes necessary.



# Module Outline (Cont.)

- **Mixed Level Modeling**
  - ○ **Mixed Level Modeling Objectives**
  - ○ **Mixed Level Modeling Approaches**
  - ○ **Mixed Level Modeling Examples**
- **Module Summary**

No notes necessary



## Module Outline

- **Performance Modeling Introduction**
  - Goals and Motivation
  - Definitions
  - Performance Modeling in the Design Process
  - Metrics
- Performance Modeling Theory
- Non VHDL-Based Performance Modeling Tools
- Techniques for Performance Modeling using VHDL
- VHDL-Based Performance Modeling Tools
- VHDL Performance Modeling Examples
- Mixed Level Modeling
- Module Summary

Module Outline

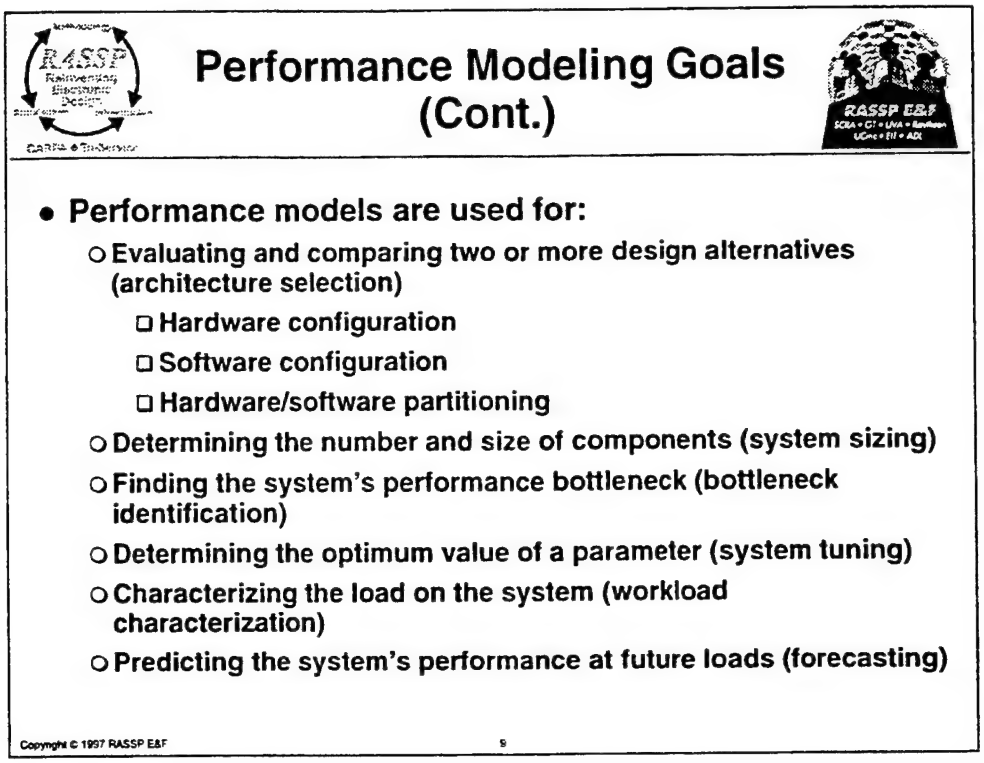
## Performance Modeling Goals

- **Estimate the performance of a given system by analyzing a high level model of the system**
  - ○ Model needs to include as little detail as necessary
    - ☐ Shorter model development time
    - ☐ Shorter model simulation time
    - ☐ Easier interpretation of the results
  - ○ Model needs to produce as accurate results as possible
    - ☐ Increasing accuracy usually means increasing detail - a conflict with the goal above
    - ☐ Performance models often may not produce accurate absolute results, but will produce accurate comparative results with a similar model of another system alternative
    - ☐ Selecting the best candidate architecture can be performed with an abstract performance model, but model must be refined to ensure performance goals are met

The goal of performance modeling is to analyze the performance model of a system using a high-level model. The model needs to be at as high (abstract) a level as possible to reduce model generation, verification, and simulation time, but at a low enough level that accurate results are obtained.

How to determine this level is not an easy process but is usually best approached from the "to little detail" side down.

Abstract performance models may not give completely accurate absolute results as in "this architecture will have a throughput of X jobs per second," but can give accurate comparative results as in "architecture A has a 20% greater throughput than architecture B."

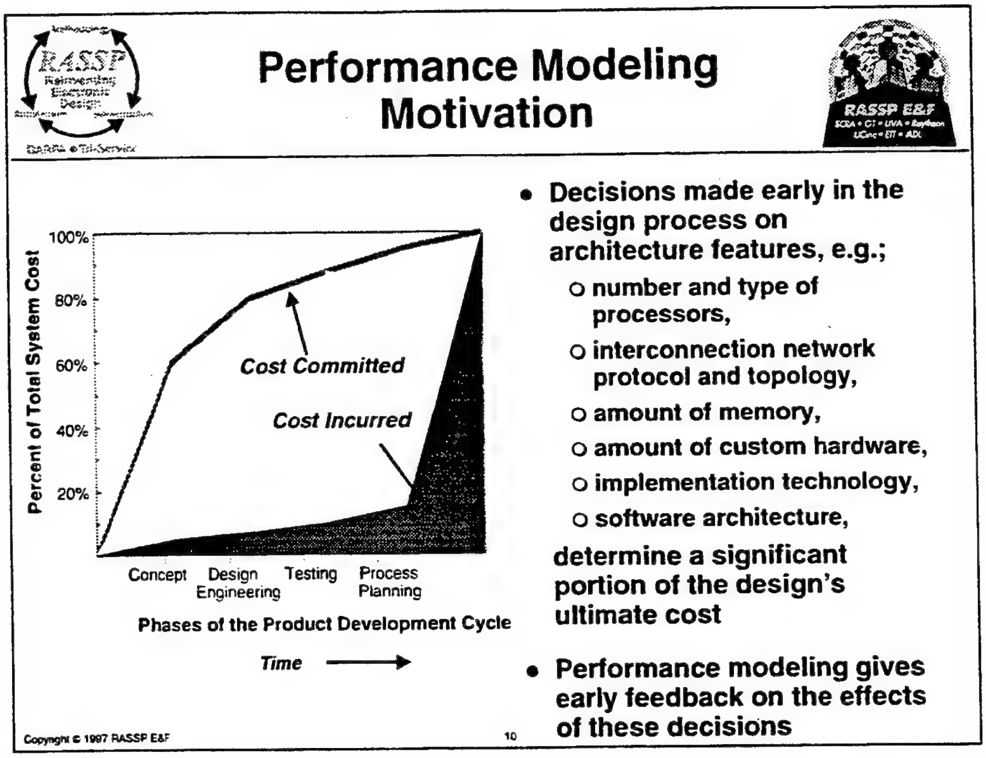


## Performance Modeling Goals (Cont.)

- Performance models are used for:
  - ○ Evaluating and comparing two or more design alternatives (architecture selection)
    - ☐ Hardware configuration
    - ☐ Software configuration
    - ☐ Hardware/software partitioning
  - ○ Determining the number and size of components (system sizing)
  - ○ Finding the system's performance bottleneck (bottleneck identification)
  - ○ Determining the optimum value of a parameter (system tuning)
  - ○ Characterizing the load on the system (workload characterization)
  - ○ Predicting the system's performance at future loads (forecasting)

This list comes from many of the references, but mainly from [Jain91].

In this module, we are discussing the mainly the application of performance modeling to the architecture selection process.



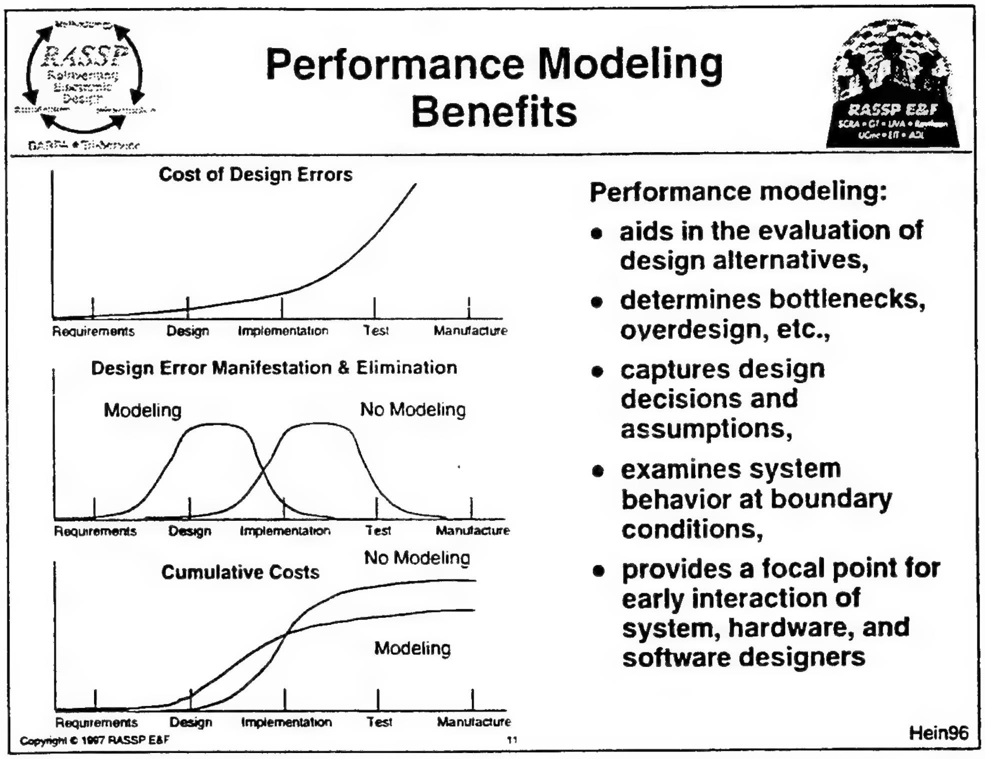
## Performance Modeling Motivation

- Decisions made early in the design process on architecture features, e.g.;
  - o number and type of processors,
  - o interconnection network protocol and topology,
  - o amount of memory,
  - o amount of custom hardware,
  - o implementation technology,
  - o software architecture,

  determine a significant portion of the design's ultimate cost
- Performance modeling gives early feedback on the effects of these decisions

This graph shows that most of the final cost of a system is locked in during the early phases of the design process when the architecture of the system is selected. However, the cost incurred in designing and producing the system does not reach its peak until the product is going out the door. Therefore, spending some time (and money) looking at the final cost of candidate architectures and their performance, early in the design process can save a great deal.

Note that these curves will change some if performance modeling is used in that more cost will be incurred early as design cost for the early stages increases, and the cost committed early will be less as the actual selection of the architecture is done later in the design cycle.


Cost of Design Errors
Requirements
Design
Implementation
Test
Manufacture
Design Error Manifestation & Elimination
Modeling
No Modeling
Requirements
Design
Implementation
Test
Manufacture
Cumulative Costs
No Modeling
Modeling
Requirements
Design
Implementation
Test
Manufacture

## Performance Modeling Benefits

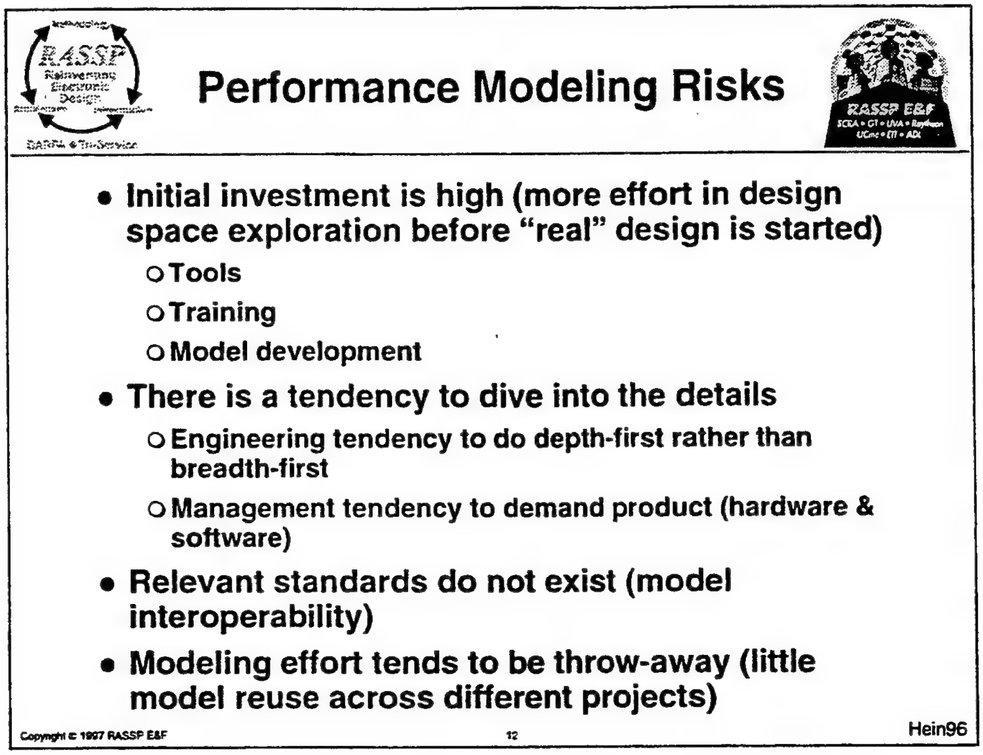
**Performance modeling:**

- **aids in the evaluation of design alternatives,**
- **determines bottlenecks, overdesign, etc.,**
- **captures design decisions and assumptions,**
- **examines system behavior at boundary conditions,**
- **provides a focal point for early interaction of system, hardware, and software designers**

Hein96

11

This slide shows some of the benefits of performance modeling as seen by some industrial users of the technique. Note that using performance modeling results in design errors being manifested and eliminated earlier in the design process where they are less costly. Also note that initially, the cost of a design process with performance modeling is higher, but the overall cost (area under the curve) is lower.

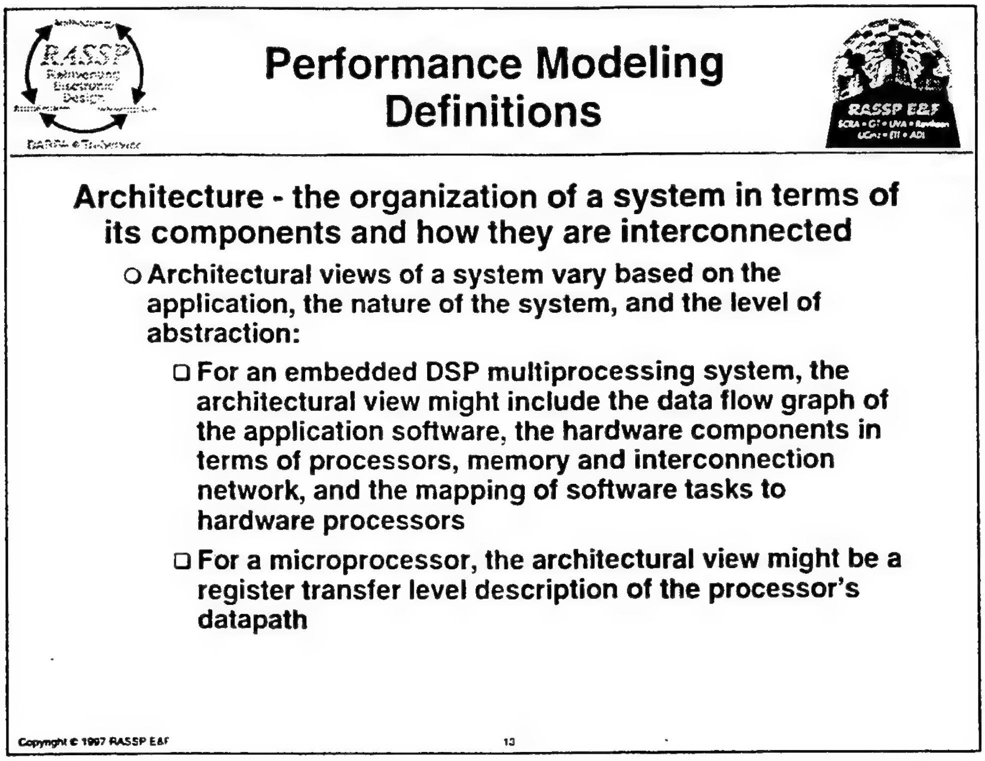


## Performance Modeling Risks

- Initial investment is high (more effort in design space exploration before "real" design is started)
  - Tools
  - Training
  - Model development
- There is a tendency to dive into the details
  - Engineering tendency to do depth-first rather than breadth-first
  - Management tendency to demand product (hardware & software)
- Relevant standards do not exist (model interoperability)
- Modeling effort tends to be throw-away (little model reuse across different projects)

Copyright © 1997 RASSP E&F 12 Hein96

The initial investment in performance modeling is high in that it increases the time spent in design space exploration before the design of the chosen architecture is actually started. This is increased by the fact that often, designers need to be trained to use the tools and develop the models necessary for performance modeling. However, the goal of performance modeling is to significantly reduce the detailed design time and cost for the chosen system by eliminating costly redesigns and design errors, thereby decreasing the overall design time and cost.

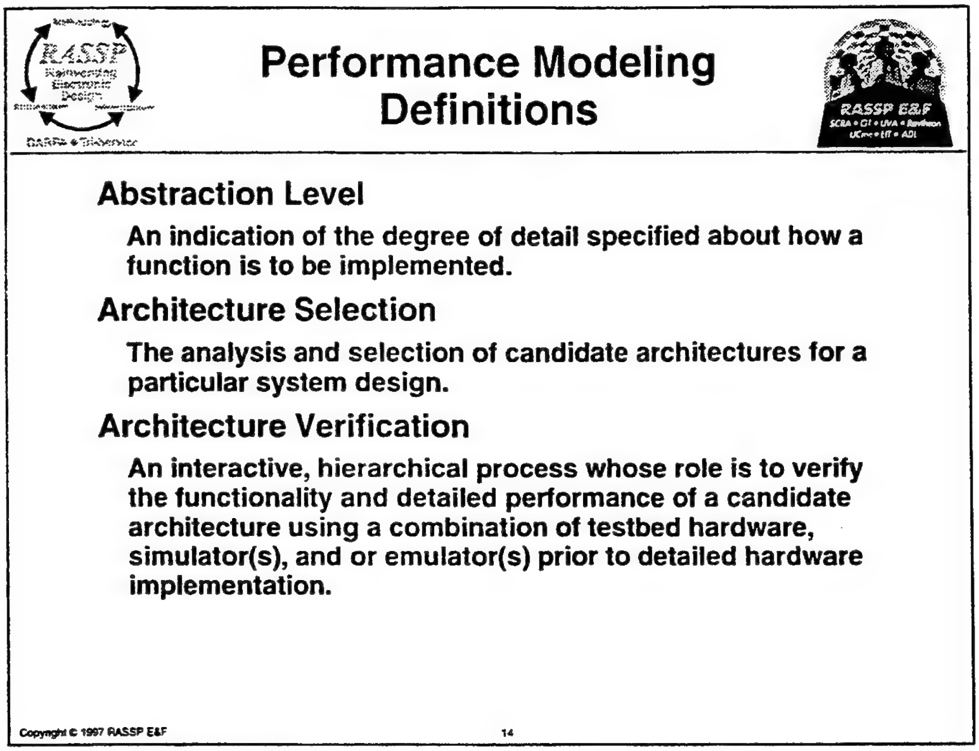


## Performance Modeling Definitions

**Architecture - the organization of a system in terms of its components and how they are interconnected**

- ○ Architectural views of a system vary based on the application, the nature of the system, and the level of abstraction:
  - ☐ For an embedded DSP multiprocessing system, the architectural view might include the data flow graph of the application software, the hardware components in terms of processors, memory and interconnection network, and the mapping of software tasks to hardware processors
  - ☐ For a microprocessor, the architectural view might be a register transfer level description of the processor's datapath

The definition of architecture is different for different systems and different levels of abstraction.



## Performance Modeling Definitions

**Abstraction Level**

An indication of the degree of detail specified about how a function is to be implemented.

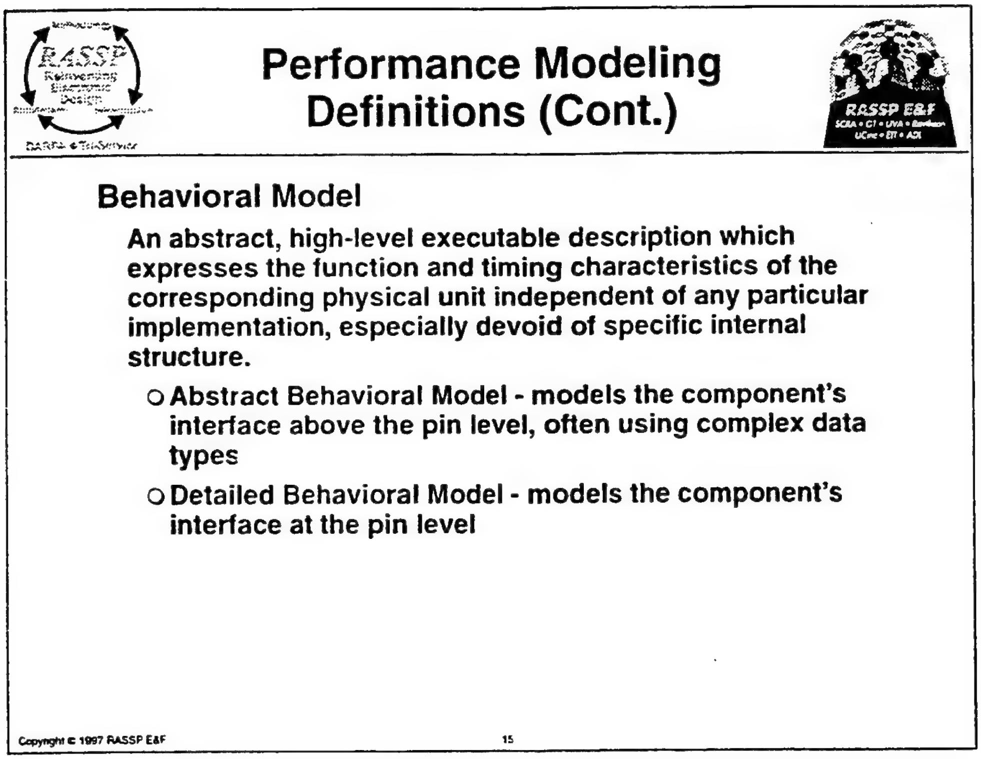
**Architecture Selection**

The analysis and selection of candidate architectures for a particular system design.

**Architecture Verification**

An interactive, hierarchical process whose role is to verify the functionality and detailed performance of a candidate architecture using a combination of testbed hardware, simulator(s), and or emulator(s) prior to detailed hardware implementation.

Architecture selection and architecture verification will be explained in more detail later in the module.



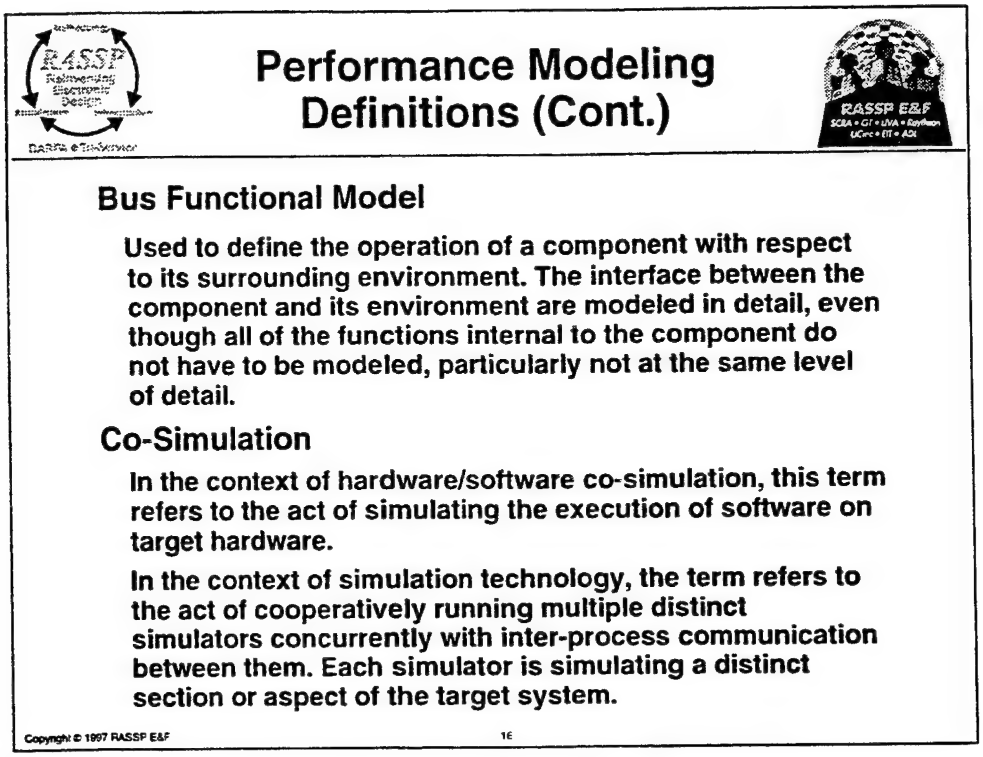
## Performance Modeling Definitions (Cont.)

**Behavioral Model**

An abstract, high-level executable description which expresses the function and timing characteristics of the corresponding physical unit independent of any particular implementation, especially devoid of specific internal structure.

- Abstract Behavioral Model - models the component's interface above the pin level, often using complex data types
- Detailed Behavioral Model - models the component's interface at the pin level

All definitions of model types are consistent with the RASSP Taxonomy [Hein97]



## Performance Modeling Definitions (Cont.)

### Bus Functional Model

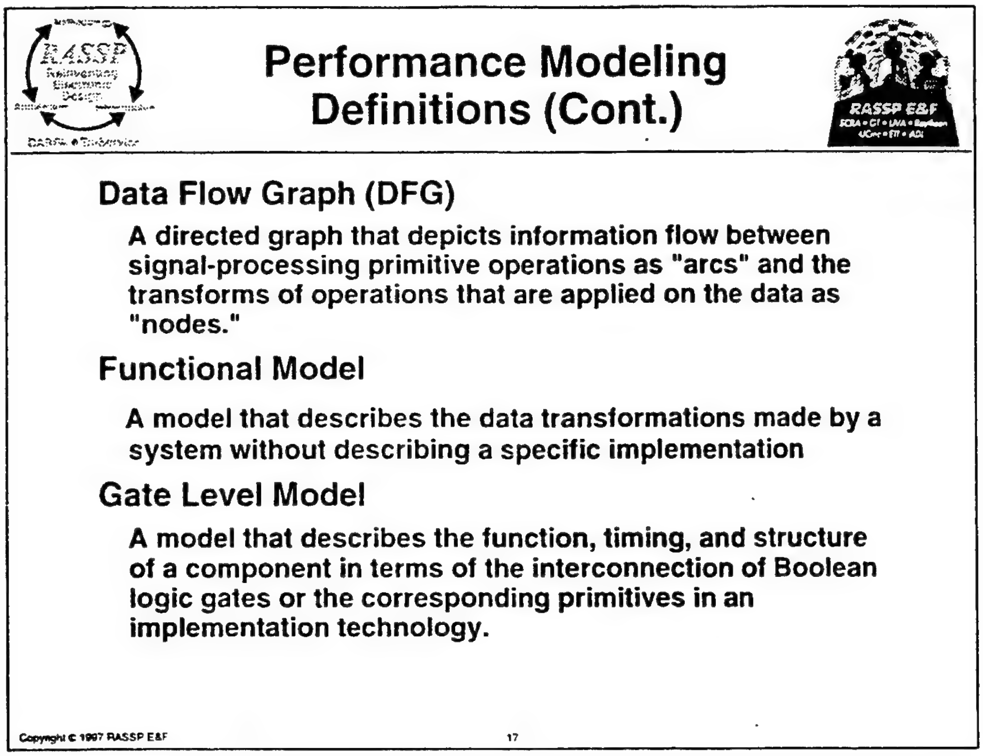
Used to define the operation of a component with respect to its surrounding environment. The interface between the component and its environment are modeled in detail, even though all of the functions internal to the component do not have to be modeled, particularly not at the same level of detail.

### Co-Simulation

In the context of hardware/software co-simulation, this term refers to the act of simulating the execution of software on target hardware.

In the context of simulation technology, the term refers to the act of cooperatively running multiple distinct simulators concurrently with inter-process communication between them. Each simulator is simulating a distinct section or aspect of the target system.

No notes necessary.



## Performance Modeling Definitions (Cont.)

**Data Flow Graph (DFG)**

A directed graph that depicts information flow between signal-processing primitive operations as "arcs" and the transforms of operations that are applied on the data as "nodes."

**Functional Model**

A model that describes the data transformations made by a system without describing a specific implementation

**Gate Level Model**

A model that describes the function, timing, and structure of a component in terms of the interconnection of Boolean logic gates or the corresponding primitives in an implementation technology.

No notes necessary.

RASSP
Reinventing Electronic Design
DARPA • Tri-Service

# Performance Modeling Definitions (Cont.)

RASSP E&F
SCRA • GT • UVA • Raytheon
UCinc • EIT • ADI

## Hardware/Software Codesign

The joint development and verification of both hardware and software via simulation/emulation from the hardware/software partitioning of functionality through design release.

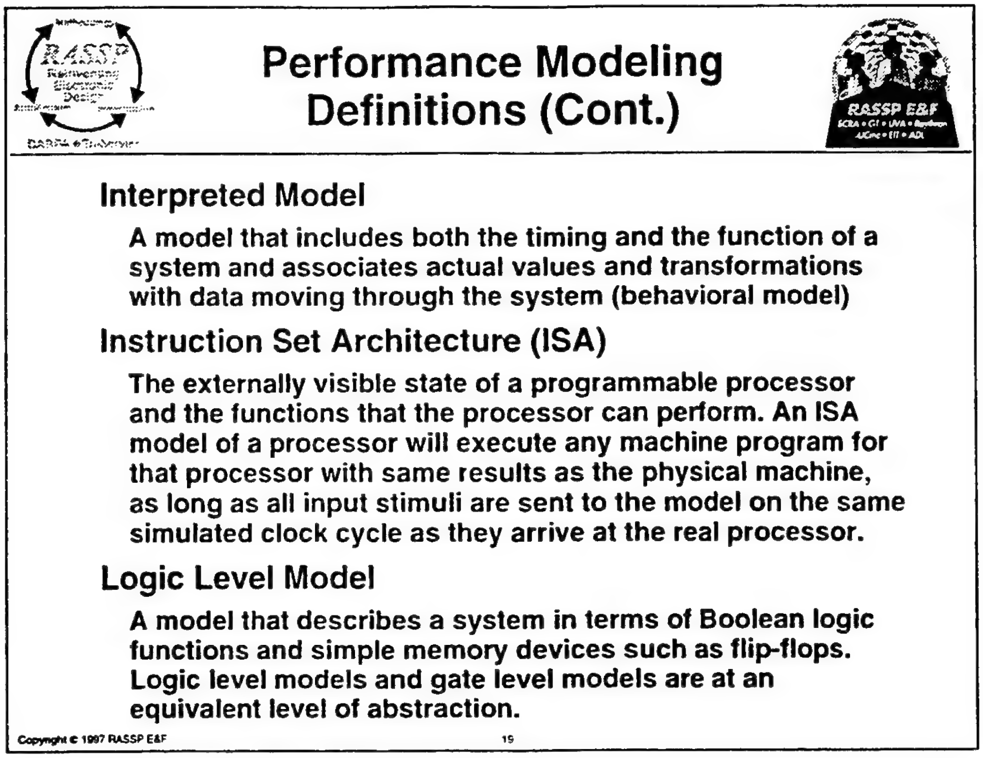
## Hierarchy

A multi-level classification system that supports aggregation of components into larger components and decomposition of components into lower level components.

## Implementation Model

A model that reflects the design of a specific physical implementation of a hardware component.

No notes necessary.



## Performance Modeling Definitions (Cont.)

**Interpreted Model**

A model that includes both the timing and the function of a system and associates actual values and transformations with data moving through the system (behavioral model)

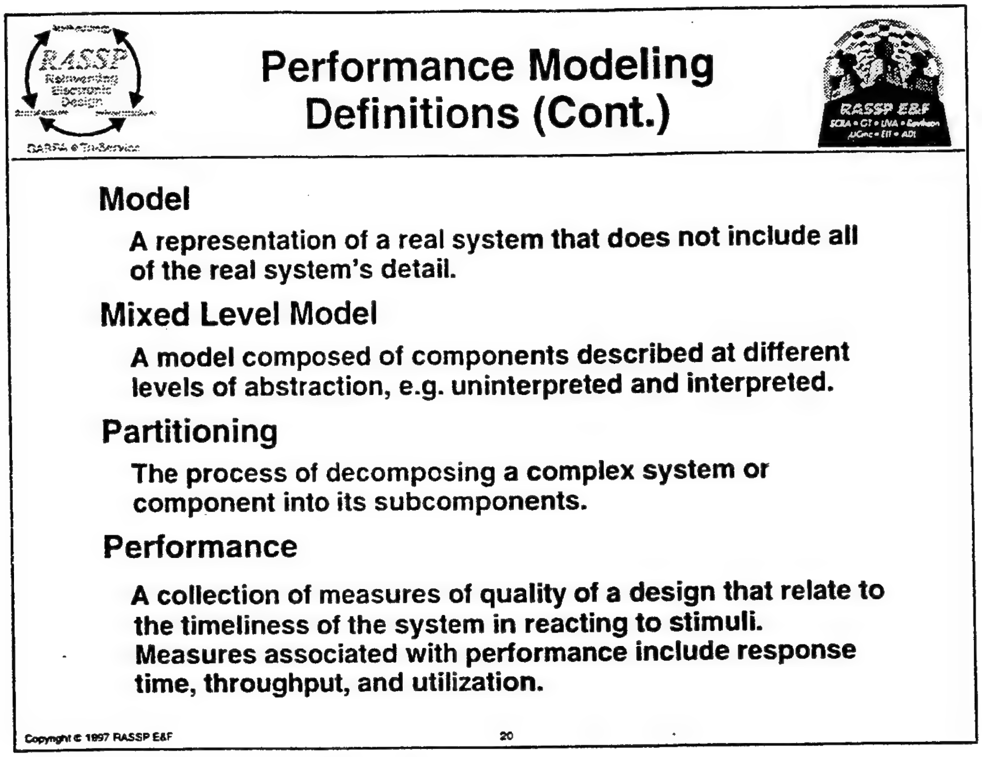
**Instruction Set Architecture (ISA)**

The externally visible state of a programmable processor and the functions that the processor can perform. An ISA model of a processor will execute any machine program for that processor with same results as the physical machine, as long as all input stimuli are sent to the model on the same simulated clock cycle as they arrive at the real processor.

**Logic Level Model**

A model that describes a system in terms of Boolean logic functions and simple memory devices such as flip-flops. Logic level models and gate level models are at an equivalent level of abstraction.

No notes necessary.



## Performance Modeling Definitions (Cont.)

**Model**

A representation of a real system that does not include all of the real system's detail.

**Mixed Level Model**

A model composed of components described at different levels of abstraction, e.g. uninterpreted and interpreted.

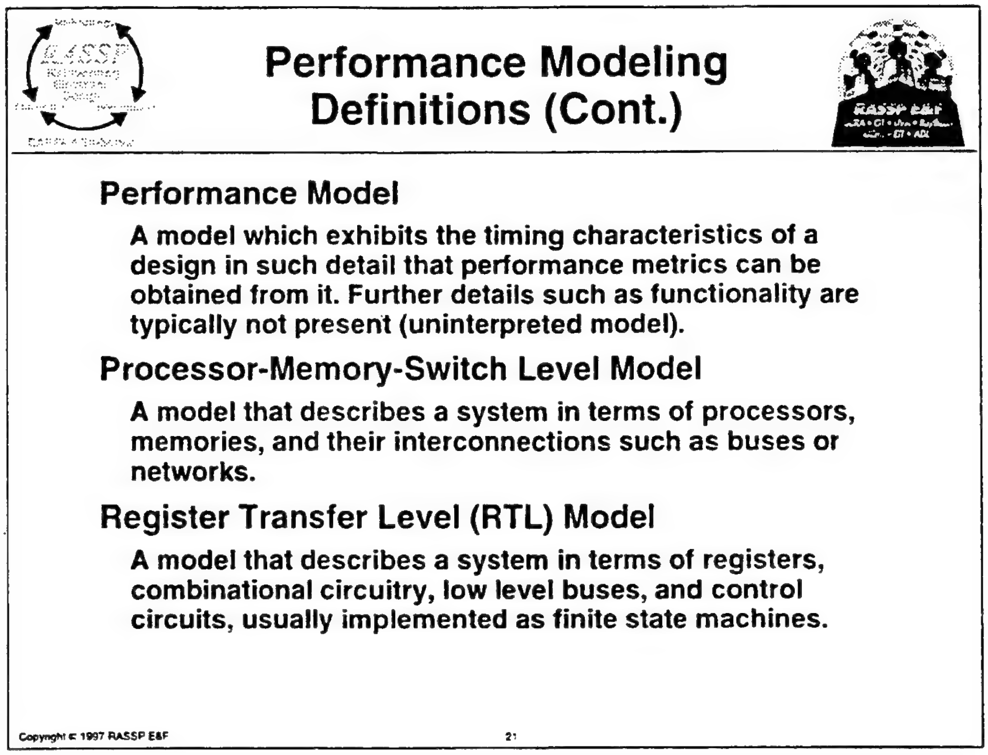
**Partitioning**

The process of decomposing a complex system or component into its subcomponents.

**Performance**

A collection of measures of quality of a design that relate to the timeliness of the system in reacting to stimuli. Measures associated with performance include response time, throughput, and utilization.

No notes necessary.



## Performance Modeling Definitions (Cont.)

**Performance Model**

A model which exhibits the timing characteristics of a design in such detail that performance metrics can be obtained from it. Further details such as functionality are typically not present (uninterpreted model).

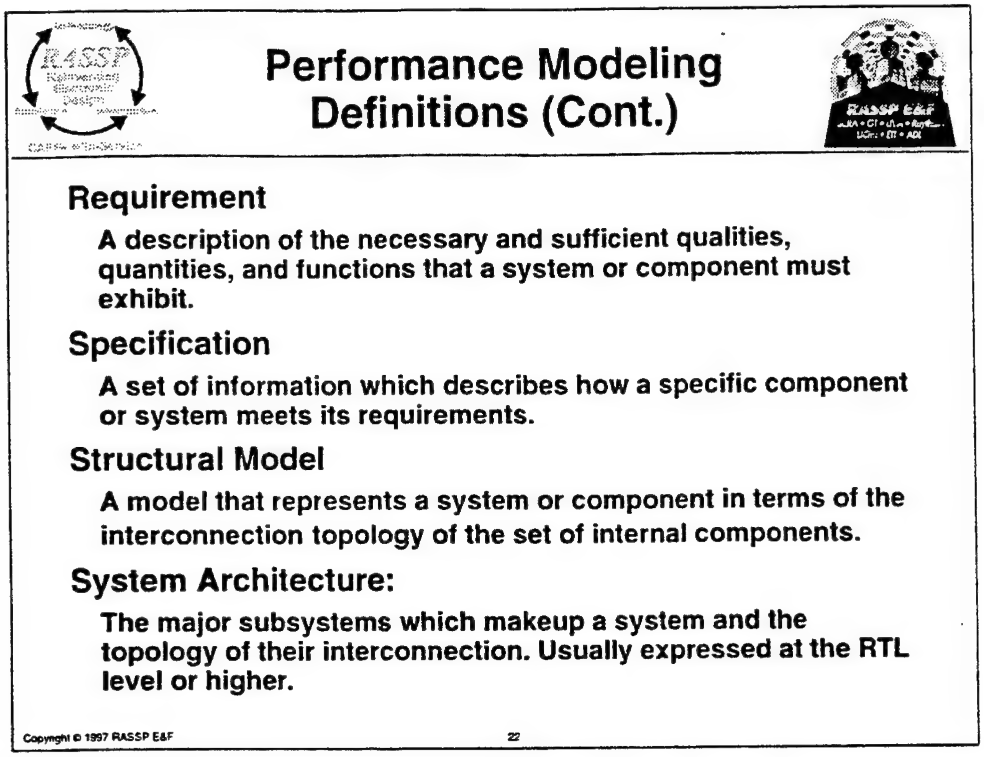
**Processor-Memory-Switch Level Model**

A model that describes a system in terms of processors, memories, and their interconnections such as buses or networks.

**Register Transfer Level (RTL) Model**

A model that describes a system in terms of registers, combinational circuitry, low level buses, and control circuits, usually implemented as finite state machines.

No notes necessary.



## Performance Modeling Definitions (Cont.)

**Requirement**

A description of the necessary and sufficient qualities, quantities, and functions that a system or component must exhibit.

**Specification**

A set of information which describes how a specific component or system meets its requirements.

**Structural Model**

A model that represents a system or component in terms of the interconnection topology of the set of internal components.

**System Architecture:**

The major subsystems which makeup a system and the topology of their interconnection. Usually expressed at the RTL level or higher.

No notes necessary.

166

# Performance Modeling Definitions (Cont.)

## System Definition

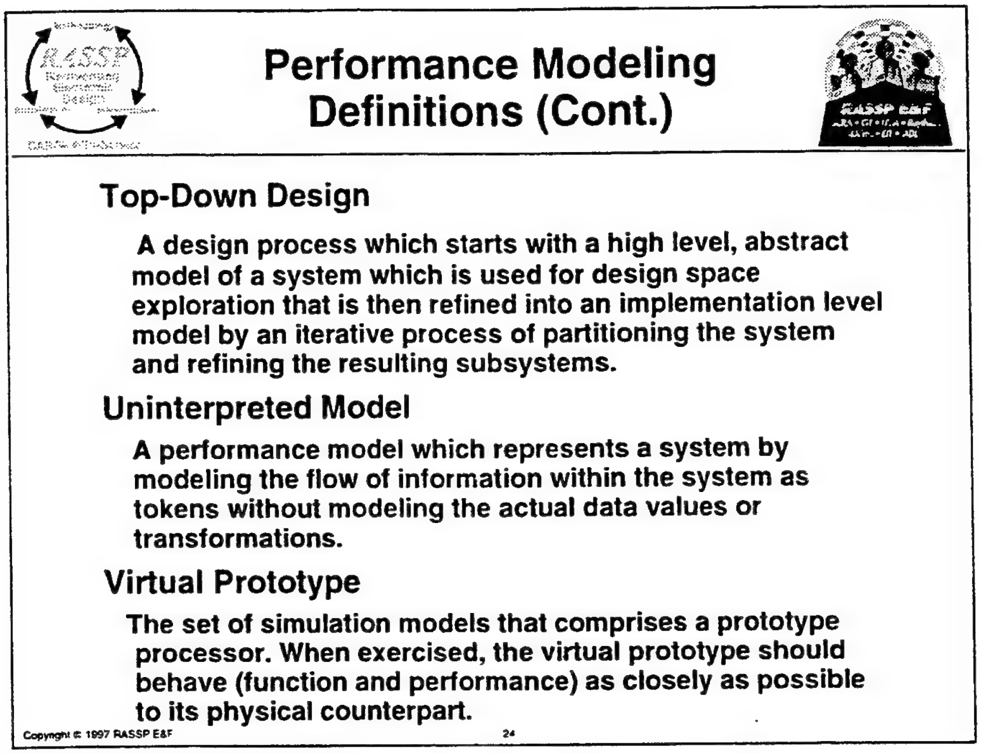
The process of analyzing customer requirements, performing functional analysis and system synthesis, and performing system level trade-offs to determine the functional and performance specifications for each subsystem.

## Token

In the context of simulation-based performance modeling, an abstract representation of a packet of data in a system. This representation may contain information about the amount of data it represents, the data's source, destination, and its route, but usually doesn't contain a representation of the data's value.

In the context of a Petri Net, a representation that the conditions described by a "place" in the Petri Net are satisfied.

No notes necessary.



## Performance Modeling Definitions (Cont.)

**Top-Down Design**

A design process which starts with a high level, abstract model of a system which is used for design space exploration that is then refined into an implementation level model by an iterative process of partitioning the system and refining the resulting subsystems.

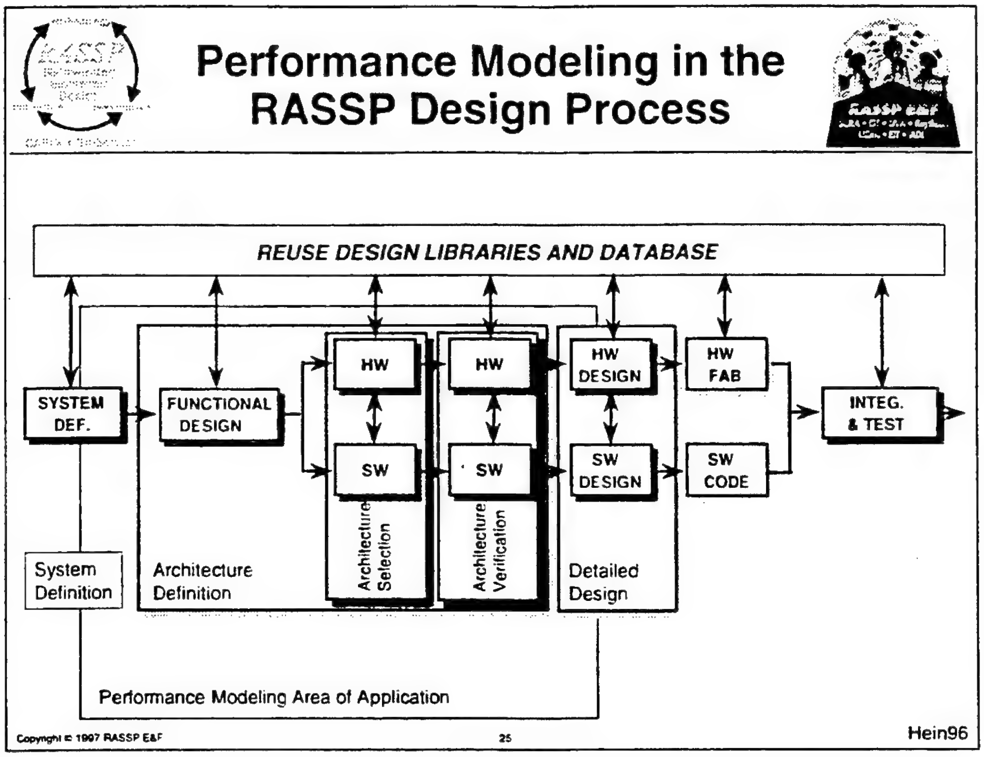
**Uninterpreted Model**

A performance model which represents a system by modeling the flow of information within the system as tokens without modeling the actual data values or transformations.

**Virtual Prototype**

The set of simulation models that comprises a prototype processor. When exercised, the virtual prototype should behave (function and performance) as closely as possible to its physical counterpart.

No notes necessary.



## Performance Modeling in the RASSP Design Process

REUSE DESIGN LIBRARIES AND DATABASE

SYSTEM DEF.

FUNCTIONAL DESIGN

HW

SW

Architecture Selection

HW

SW

Architecture Verification

HW DESIGN

SW DESIGN

HW FAB

SW CODE

INTEG. & TEST

System Definition

Architecture Definition

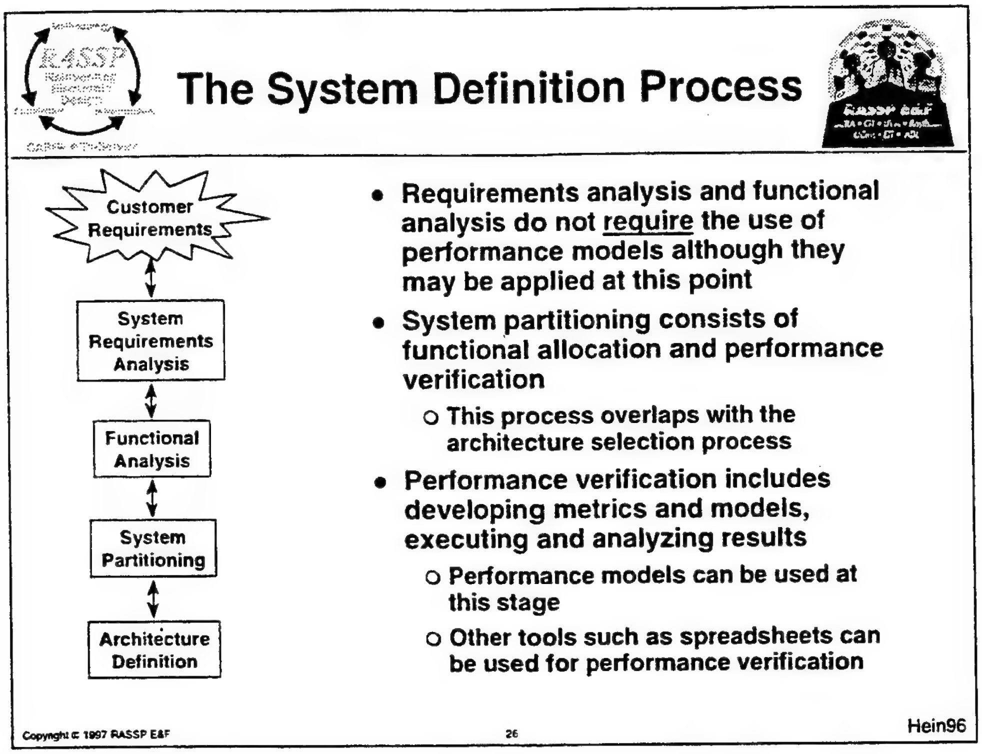
Detailed Design

Performance Modeling Area of Application

Hein96

This slide shows the RASSP (Rapid Prototyping of Application Specific Signal Processors) design process and where performance modeling fits into it. This includes the processes of System Definition, Architecture Definition, and portions of Detailed Design. Note that Architecture Definition encompasses Functional Design and the processes of Architectural Selection Architectural Verification.

How performance modeling is used within these processes is covered in the following slides...



## The System Definition Process

Customer Requirements

System Requirements Analysis

Functional Analysis

System Partitioning

Architecture Definition

- Requirements analysis and functional analysis do not <u>require</u> the use of performance models although they may be applied at this point
- System partitioning consists of functional allocation and performance verification
  - ○ This process overlaps with the architecture selection process
- Performance verification includes developing metrics and models, executing and analyzing results
  - ○ Performance models can be used at this stage
  - ○ Other tools such as spreadsheets can be used for performance verification

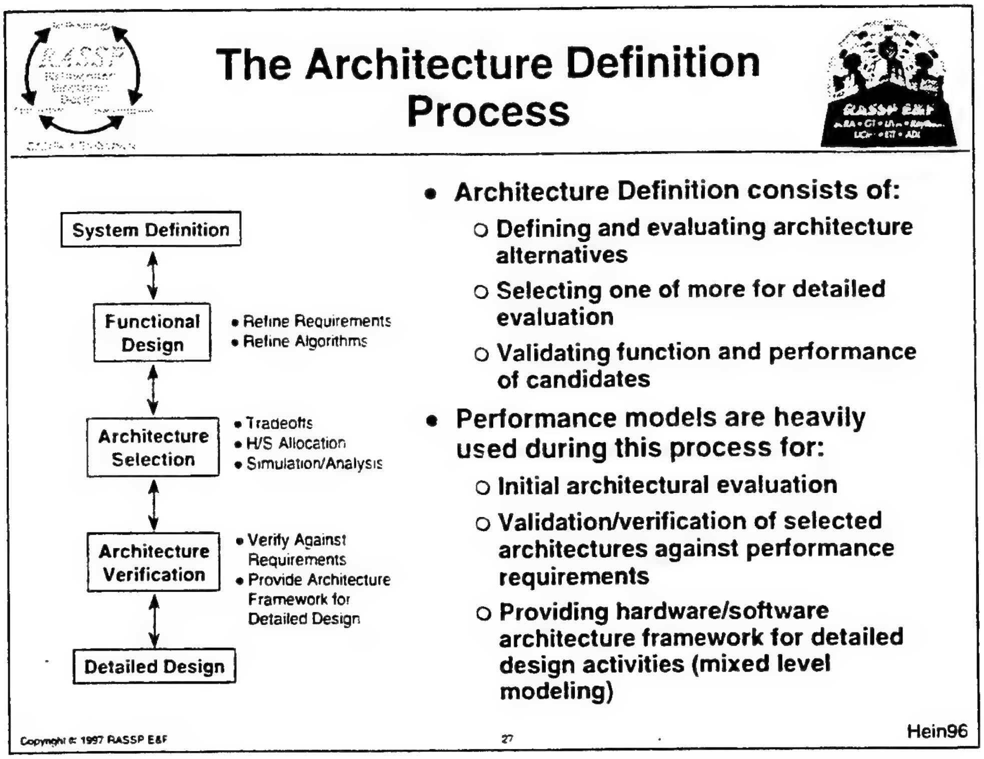
Copyright © 1997 RASSP E&F

26

Hein96

This slide presents the functions in the system definition process, which begins with customer requirements (which may be executable) and flows into the architecture definition process.

Performance models are not required at the upper levels of the system definition process although they can be applied at any point. In the performance verification phase, some type of performance modeling is required for all but the most trivial of systems.



## The Architecture Definition Process

System Definition
↕
Functional Design
- Refine Requirements
- Refine Algorithms

↕
Architecture Selection
- Tradeoffs
- H/S Allocation
- Simulation/Analysis

↕
Architecture Verification
- Verify Against Requirements
- Provide Architecture Framework for Detailed Design

↕
Detailed Design

- Architecture Definition consists of:
  - Defining and evaluating architecture alternatives
  - Selecting one of more for detailed evaluation
  - Validating function and performance of candidates
- Performance models are heavily used during this process for:
  - Initial architectural evaluation
  - Validation/verification of selected architectures against performance requirements
  - Providing hardware/software architecture framework for detailed design activities (mixed level modeling)

Copyright © 1997 RASSP E&F

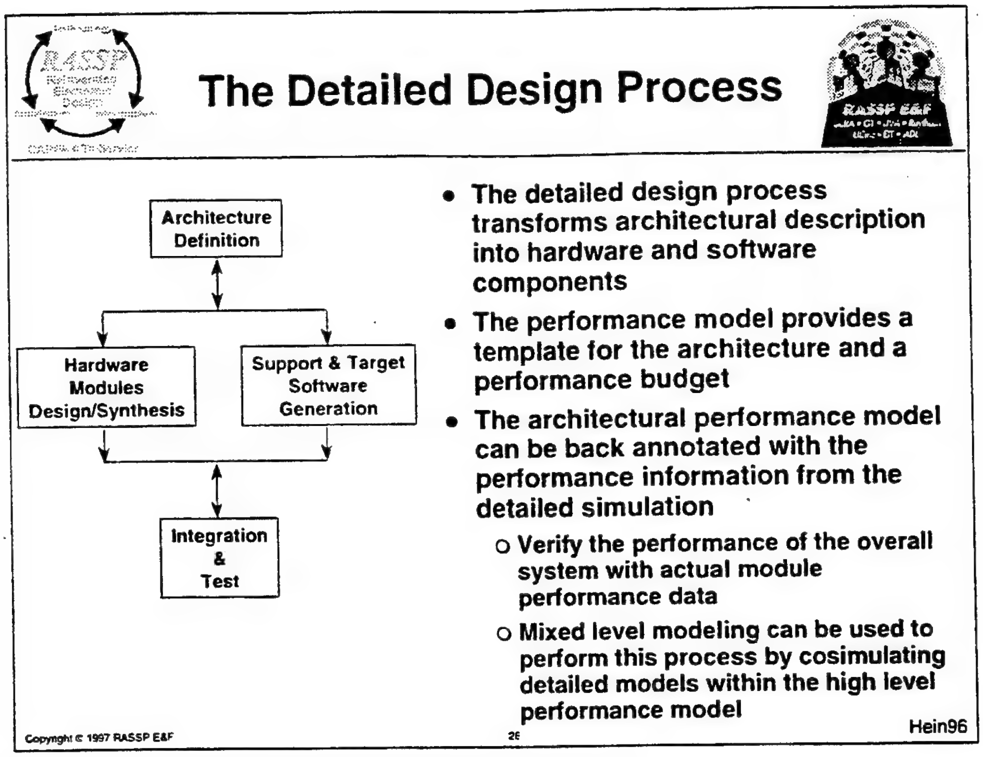
27

Hein96

The architecture definition process is fed by the system definition process and in turn feeds into the detailed design process.

Performance models can be used in the functional design process to help refine requirements and algorithms. They most definitely are used in the architecture selection and verification process for evaluation.

Note that this slide show one view of the architecture definition process, but it can be pursued in other ways (more of an iterative process, less of a waterfall, etc.)



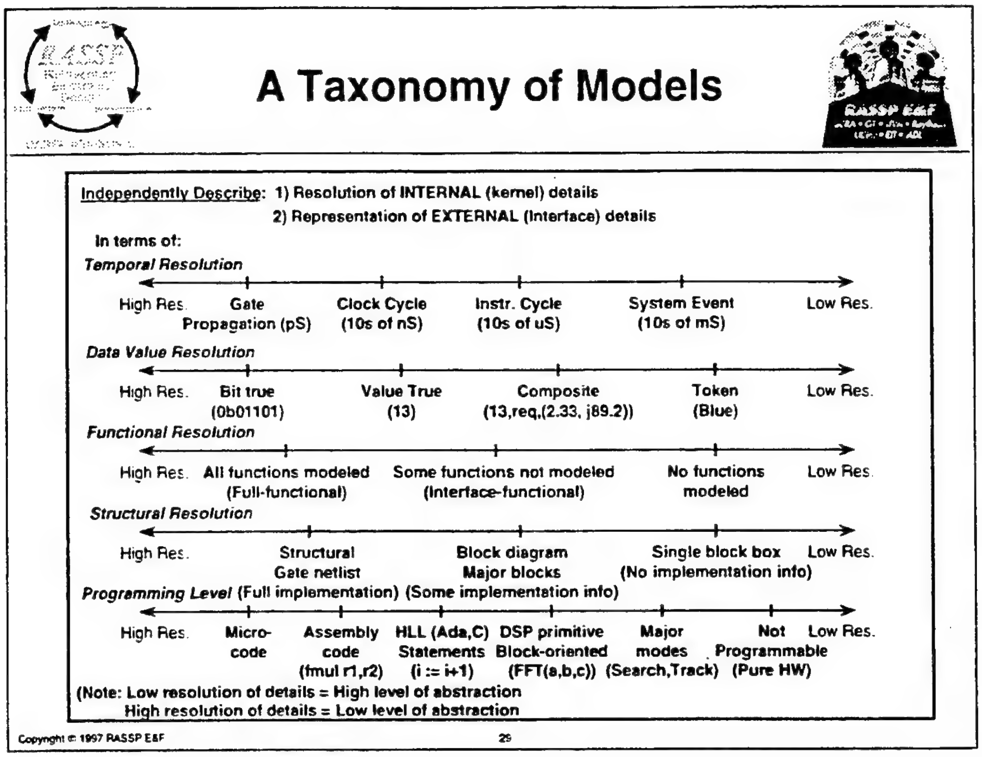
## The Detailed Design Process

Architecture Definition
Hardware Modules Design/Synthesis
Support & Target Software Generation
Integration & Test

- The detailed design process transforms architectural description into hardware and software components
- The performance model provides a template for the architecture and a performance budget
- The architectural performance model can be back annotated with the performance information from the detailed simulation
  - o Verify the performance of the overall system with actual module performance data
  - o Mixed level modeling can be used to perform this process by cosimulating detailed models within the high level performance model

Hein96

This slide shows the detailed design process and how performance modeling is used in it. Note that this is where mixed level modeling, the notion of cosimulating performance and behavioral models, is introduced.



# A Taxonomy of Models

<u>Independently Describe</u>: 1) Resolution of INTERNAL (kernel) details
2) Representation of EXTERNAL (Interface) details

In terms of:

***Temporal Resolution***

| | | | | | |
|---|---|---|---|---|---|
| High Res. | Gate<br>Propagation (pS) | Clock Cycle<br>(10s of nS) | Instr. Cycle<br>(10s of uS) | System Event<br>(10s of mS) | Low Res. |

***Data Value Resolution***

| | | | | | |
|---|---|---|---|---|---|
| High Res. | Bit true<br>(0b01101) | Value True<br>(13) | Composite<br>(13,req,(2.33, j89.2)) | Token<br>(Blue) | Low Res. |

***Functional Resolution***

| | | | | |
|---|---|---|---|---|
| High Res. | All functions modeled<br>(Full-functional) | Some functions not modeled<br>(Interface-functional) | No functions<br>modeled | Low Res. |

***Structural Resolution***

| | | | | |
|---|---|---|---|---|
| High Res. | Structural<br>Gate netlist | Block diagram<br>Major blocks | Single block box<br>(No implementation info) | Low Res. |

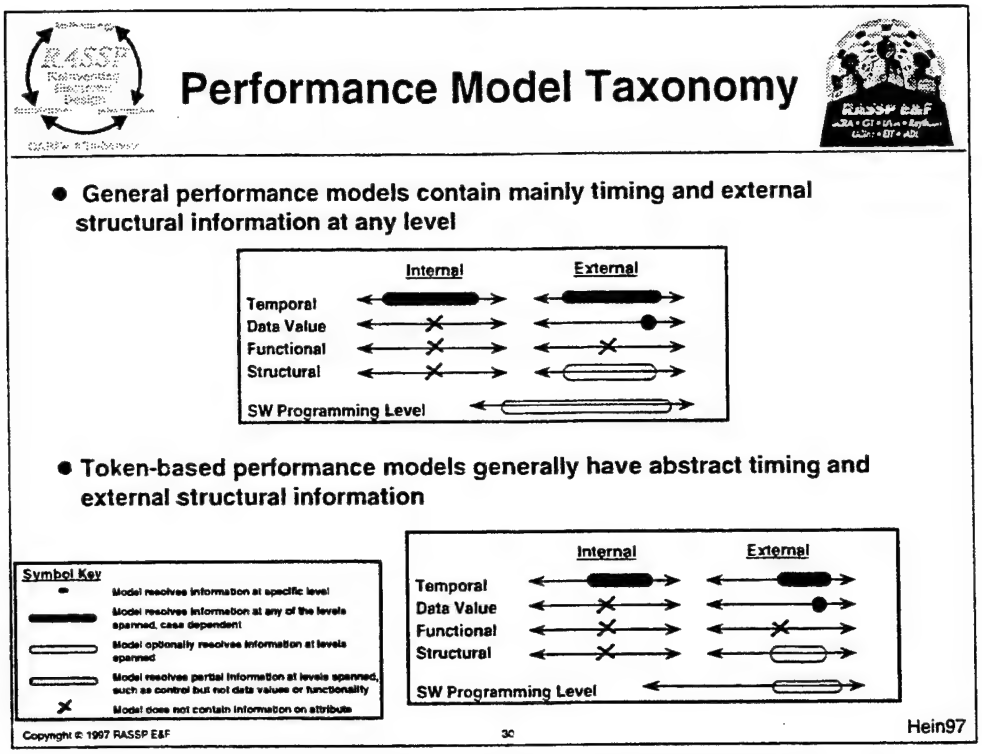
***Programming Level*** (Full implementation) (Some implementation info)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| High Res. | Micro-<br>code | Assembly<br>code<br>(fmul r1,r2) | HLL (Ada,C)<br>Statements<br>(i := i+1) | DSP primitive<br>Block-oriented<br>(FFT(a,b,c)) | Major<br>modes<br>(Search,Track) | Not<br>Programmable<br>(Pure HW) | Low Res. |

(Note: Low resolution of details = High level of abstraction
High resolution of details = Low level of abstraction

This slide shows the 5 elements of model characteristics that determines its place in the overall taxonomy of models. The position on each scale that a model occupies determines what type of model it is classified as.

This slide is taken from the RASSP taxonomy document.



## Performance Model Taxonomy

- General performance models contain mainly timing and external structural information at any level

Internal
External
Temporal
Data Value
Functional
Structural
SW Programming Level

- Token-based performance models generally have abstract timing and external structural information

Symbol Key
Model resolves information at specific level
Model resolves information at any of the levels spanned, case dependent
Model optionally resolves information at levels spanned
Model resolves partial information at levels spanned, such as control but not data values or functionality
Model does not contain information on attribute

Internal
External
Temporal
Data Value
Functional
Structural
SW Programming Level

Copyright © 1997 RASSP E&F
30
Hein97

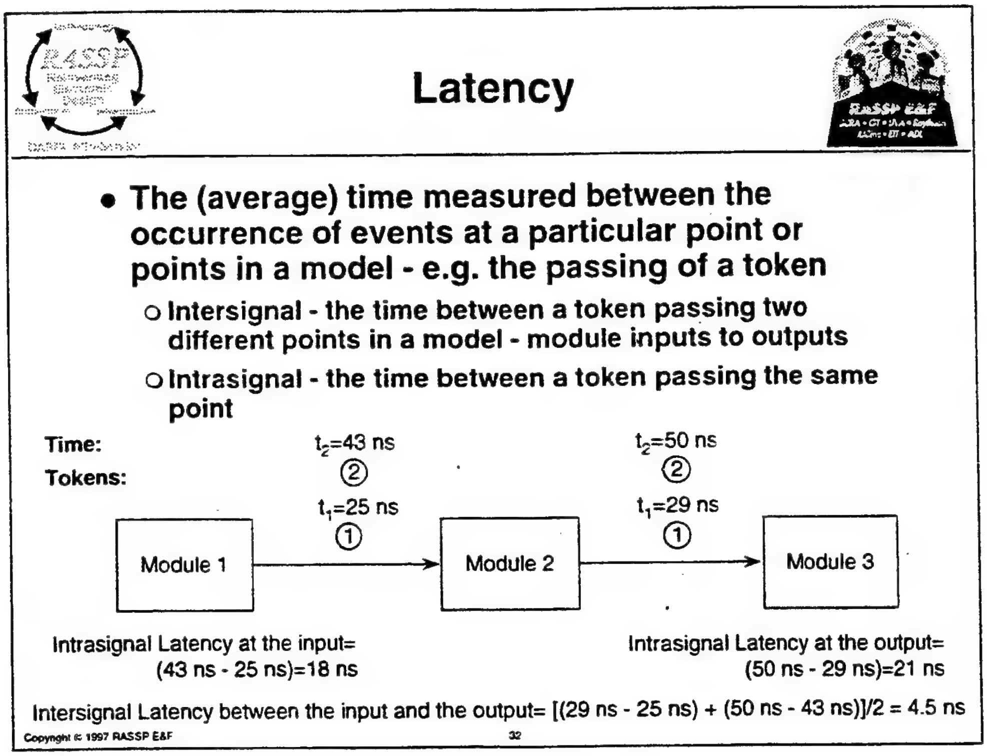
General performance models have temporal data (both internal and external) that can be at essentially any level of abstraction. They have no internal data value information, and only high level external data value information (e.g. memory address, size, etc.), no functional information and only external structural information. Software can be represented at any level.

Token-based performance models have higher levels of timing information (e.g. at the task level or data packet level, not instruction level or individual word level), and higher levels of external structure. Software is represented at the task level and above.

# Performance Modeling Metrics

- **The most common performance metrics measured from an individual performance model are:**
  - ○ **Latency**
  - ○ **Throughput**
  - ○ **Utilization**
  - ○ **Response Time**
- **Often it is desirable to study how these metrics vary with system attributes such as:**
  - ○ **Number of processors**
  - ○ **Memory size**
  - ○ **Interconnection bandwidth**
  - ○ **Clock speed**

This section will present the classical performance modeling metrics of latency, throughput, utilization, and others.



## Latency

- **The (average) time measured between the occurrence of events at a particular point or points in a model - e.g. the passing of a token**
  - **Intersignal - the time between a token passing two different points in a model - module inputs to outputs**
  - **Intrasignal - the time between a token passing the same point**

**Time:** $t_2$=43 ns — $t_2$=50 ns

**Tokens:** ② ①

$t_1$=25 ns — $t_1$=29 ns

Module 1 → Module 2 → Module 3

Intrasignal Latency at the input= (43 ns - 25 ns)=18 ns

Intrasignal Latency at the output= (50 ns - 29 ns)=21 ns

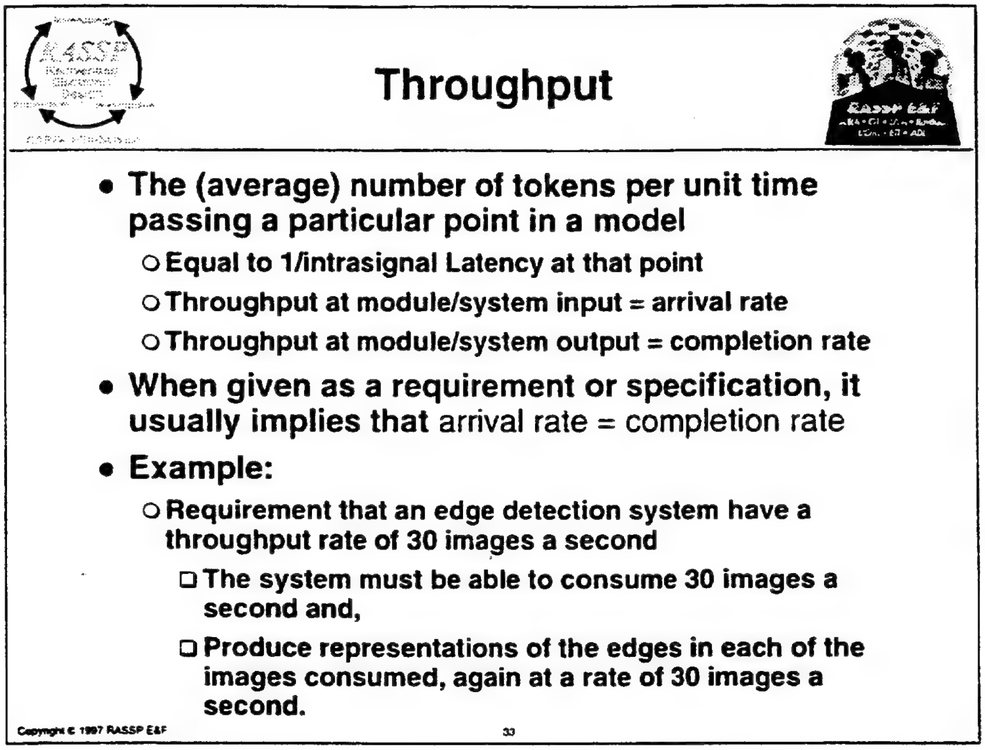
Intersignal Latency between the input and the output= [(29 ns - 25 ns) + (50 ns - 43 ns)]/2 = 4.5 ns

Copyright © 1997 RASSP E&F 32

Latency is the time between two events.

Usually, latency is the time between two events on different signals, or in different parts of the model, e.g., the time between the arrival of a memory request and a memory access - memory latency, or the time between the sending and receiving of a message - communications latency. For lack of a better term, this is called intersignal latency.

Sometimes however, the latency between events on the same signal is important, e.g., the time between subsequent memory accesses or the time between the processing of RADAR pulses by a SAR system. This is termed intrasignal latency.



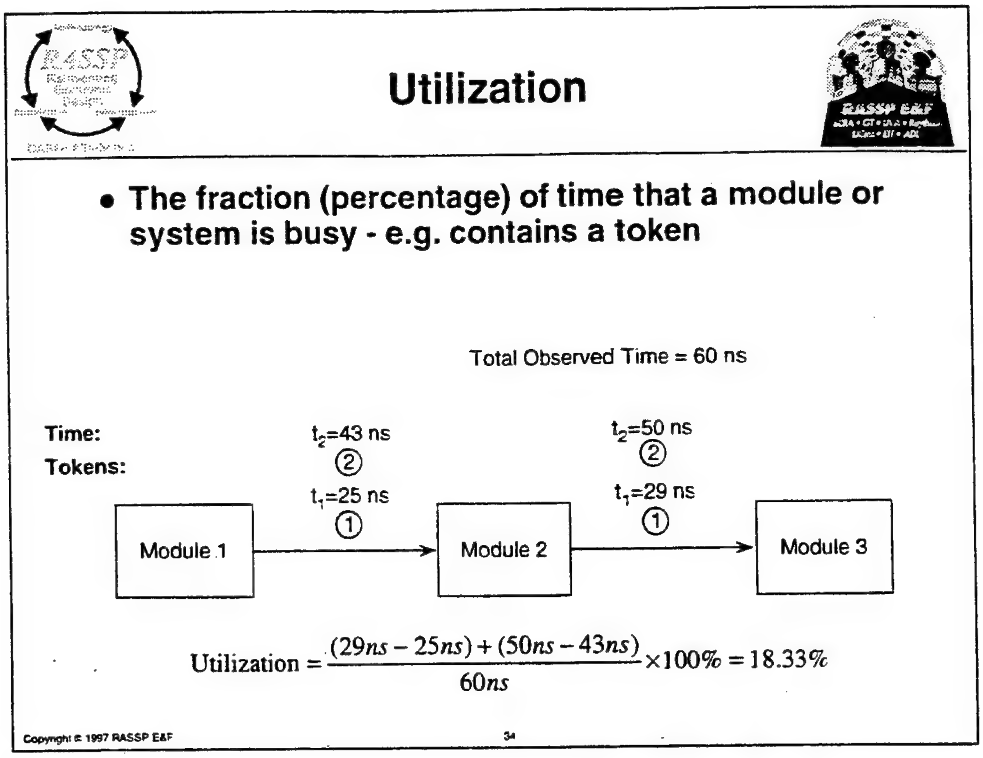
### Throughput

- **The (average) number of tokens per unit time passing a particular point in a model**
  - **Equal to 1/intrasignal Latency at that point**
  - **Throughput at module/system input = arrival rate**
  - **Throughput at module/system output = completion rate**
- **When given as a requirement or specification, it usually implies that** arrival rate = completion rate
- **Example:**
  - **Requirement that an edge detection system have a throughput rate of 30 images a second**
    - ☐ **The system must be able to consume 30 images a second and,**
    - ☐ **Produce representations of the edges in each of the images consumed, again at a rate of 30 images a second.**

Throughput is basically 1/some type of latency.

Arrival rate is 1/ the intrasignal latency at the system's input, Completion rate is 1/ the intrasignal latency at the system's output.

When used as a requirement, throughput usually means completion rate.



## Utilization

- **The fraction (percentage) of time that a module or system is busy - e.g. contains a token**

Total Observed Time = 60 ns

**Time:**
**Tokens:**

$t_2$=43 ns ②
$t_1$=25 ns ①

$t_2$=50 ns ②
$t_1$=29 ns ①

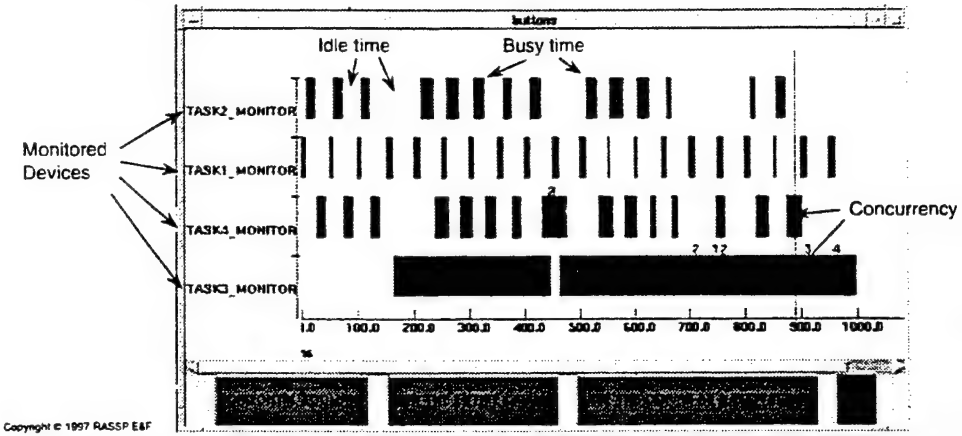
Module 1 → Module 2 → Module 3

$$\text{Utilization} = \frac{(29ns - 25ns) + (50ns - 43ns)}{60ns} \times 100\% = 18.33\%$$

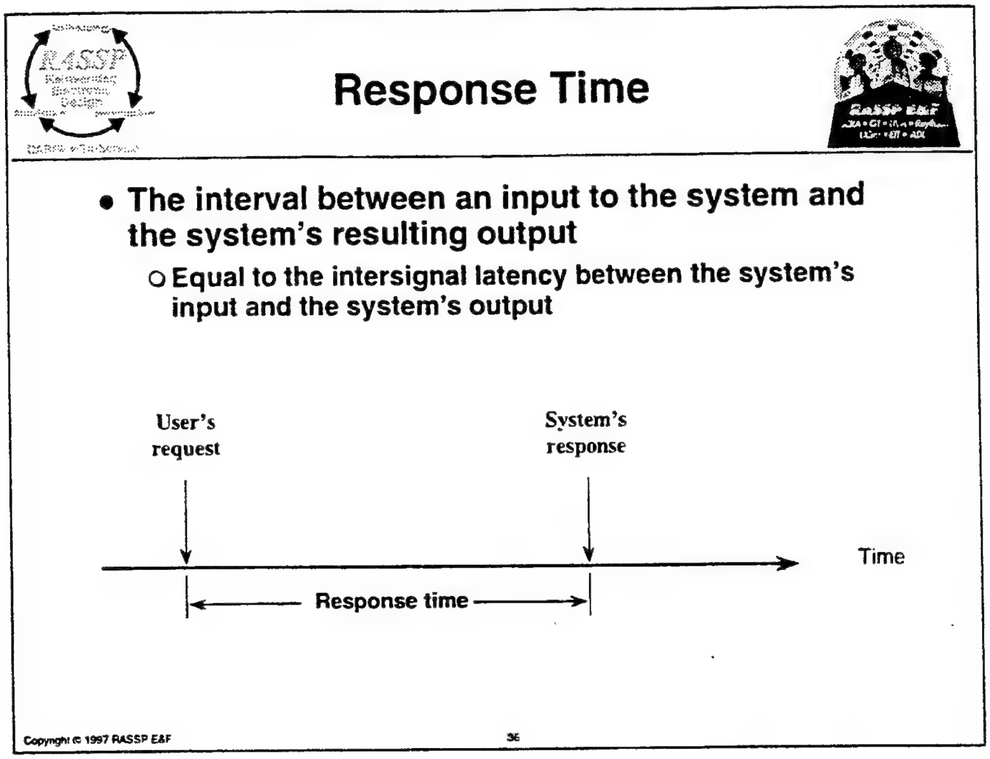
Utilization is simply the percentage of time (that the system is simulated for) that the system is actually busy i.e., it contains a token.

## Utilization (Cont.)

- **Activity Time Lines**
  - ○ **Display individual device utilization as horizontal bar graphs**
  - ○ **Useful in visualizing idle time and concurrency**



Activity time lines are a helpful way to visualize utilization, especially in a system where some concurrency is possible because they allow that concurrency to be visualized. This helps to see points where concurrency is or isn't happening

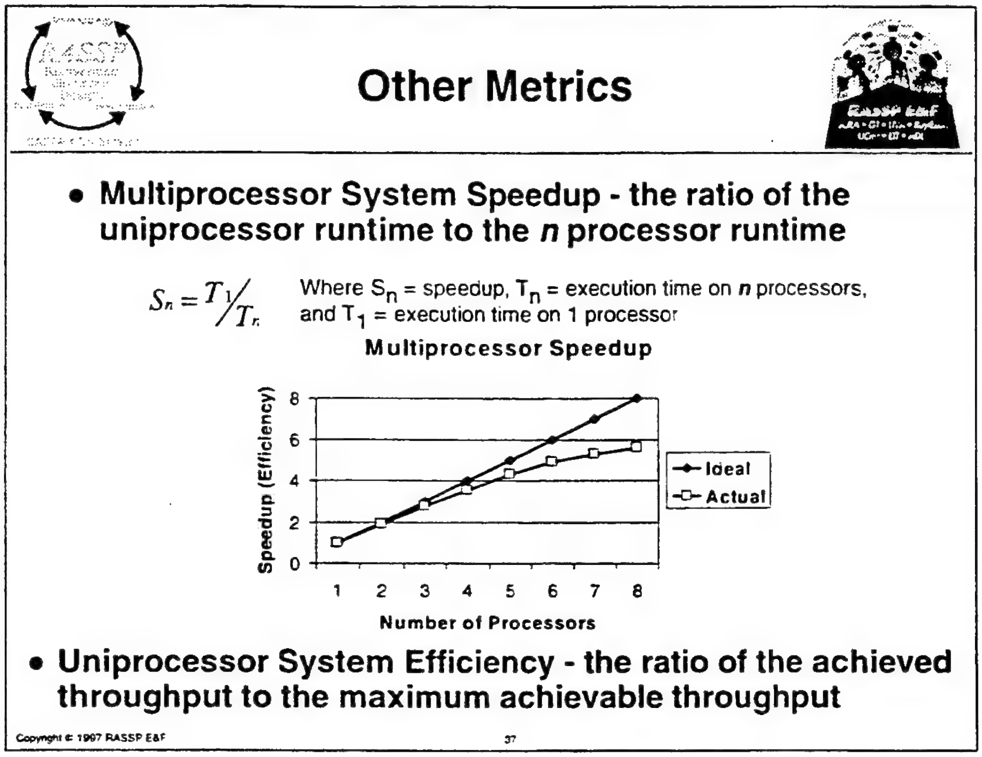


## Response Time

- **The interval between an input to the system and the system's resulting output**
  - **Equal to the intersignal latency between the system's input and the system's output**

User's request
System's response
Time
Response time

Response time is a metric that is sometimes used in "user driven systems" because it measures how long the user must wait from their input to the desired output.



## Other Metrics

- **Multiprocessor System Speedup - the ratio of the uniprocessor runtime to the *n* processor runtime**

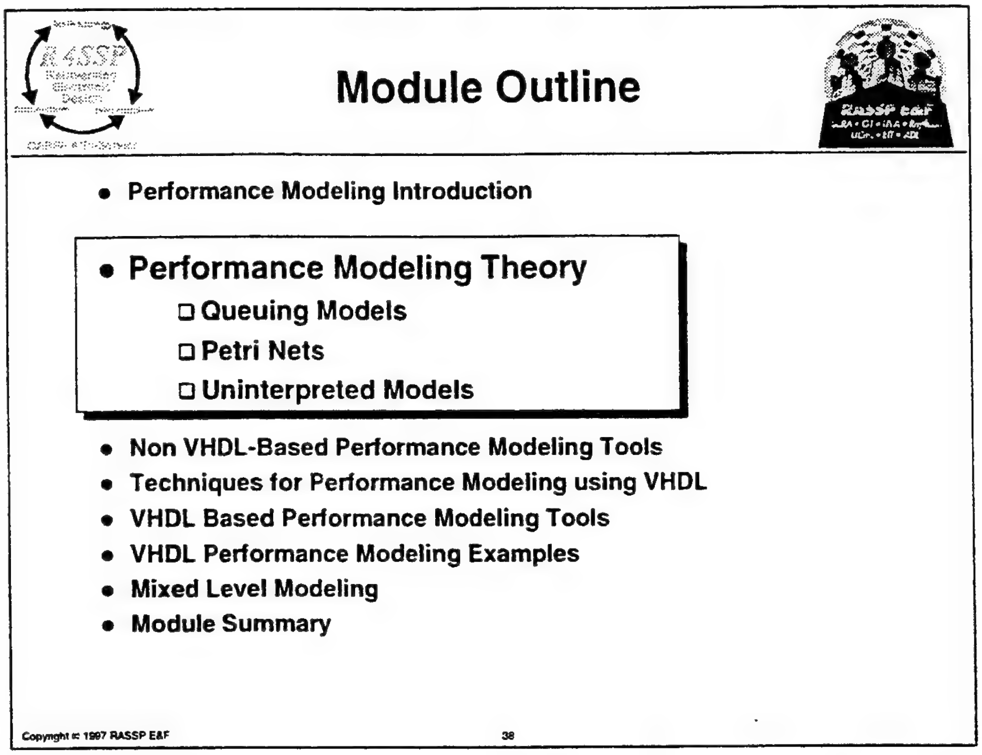
$S_n = T_1 / T_n$ Where $S_n$ = speedup, $T_n$ = execution time on *n* processors, and $T_1$ = execution time on 1 processor

**Multiprocessor Speedup**

Speedup (Efficiency)
8
6
4
2
0
1
2
3
4
5
6
7
8
Number of Processors
Ideal
Actual

- **Uniprocessor System Efficiency - the ratio of the achieved throughput to the maximum achievable throughput**

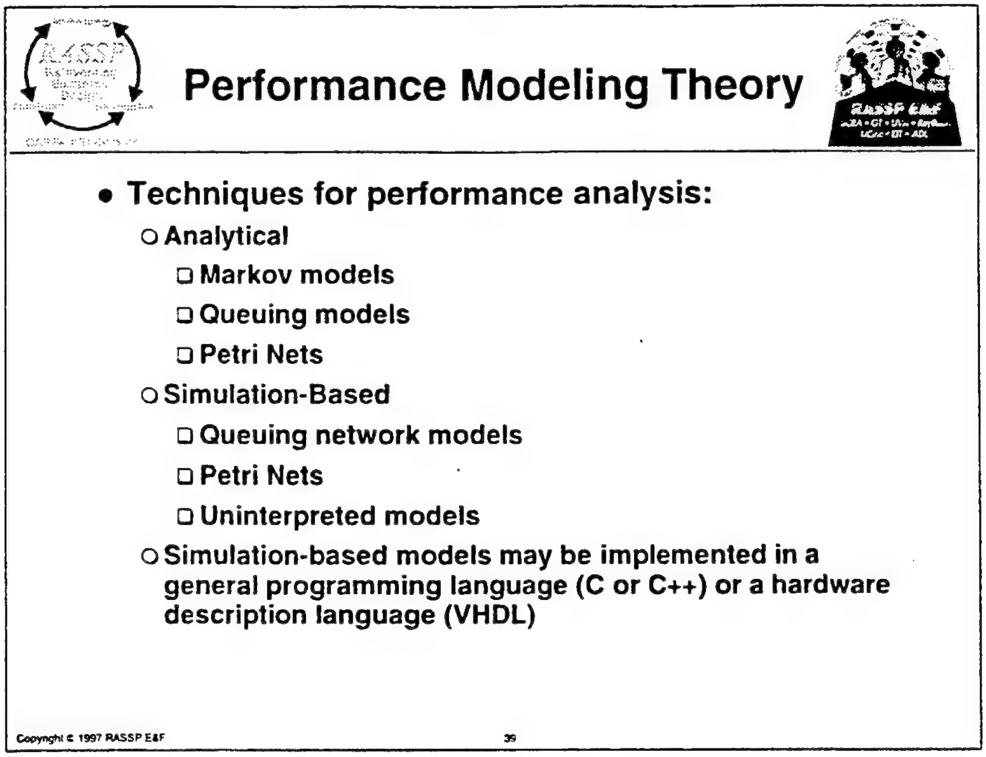
Other metrics typically used in system performance analysis include speedup for a multiprocessor system, and efficiency for a uniprocessor system (these two are related in that they are both basically the ratio of the achieved throughput to the theoretical maximum throughput, or visa versa).



## Module Outline

- Performance Modeling Introduction
- **Performance Modeling Theory**
  - ☐ Queuing Models
  - ☐ Petri Nets
  - ☐ Uninterpreted Models
- Non VHDL-Based Performance Modeling Tools
- Techniques for Performance Modeling using VHDL
- VHDL Based Performance Modeling Tools
- VHDL Performance Modeling Examples
- Mixed Level Modeling
- Module Summary

Module Outline

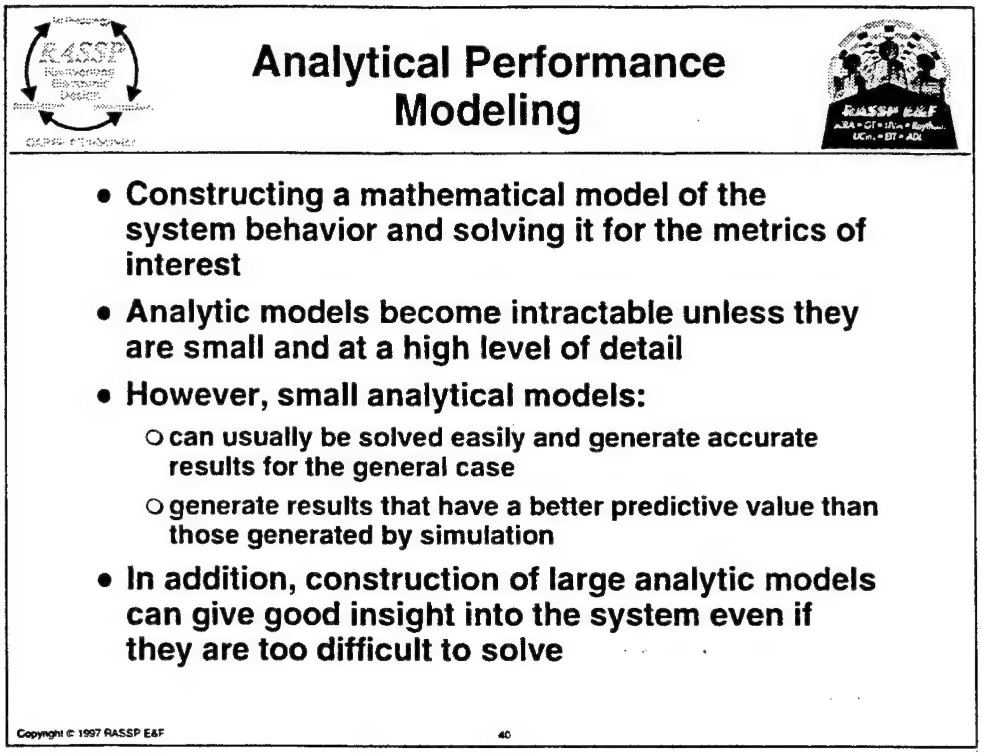


## Performance Modeling Theory

- **Techniques for performance analysis:**
  - ○ **Analytical**
    - ❑ **Markov models**
    - ❑ **Queuing models**
    - ❑ **Petri Nets**
  - ○ **Simulation-Based**
    - ❑ **Queuing network models**
    - ❑ **Petri Nets**
    - ❑ **Uninterpreted models**
  - ○ **Simulation-based models may be implemented in a general programming language (C or C++) or a hardware description language (VHDL)**

There are two basic techniques for performance modeling, analytical, and simulation-based. The advantages and disadvantages of each will be explained in each section.

Token-based performance modeling using VHDL is a simulation-based technique, but the analytical techniques will be introduced here to provide background for the simulation-based techniques. This section of the module can be omitted from discussion if this background material is not required for the given audience.

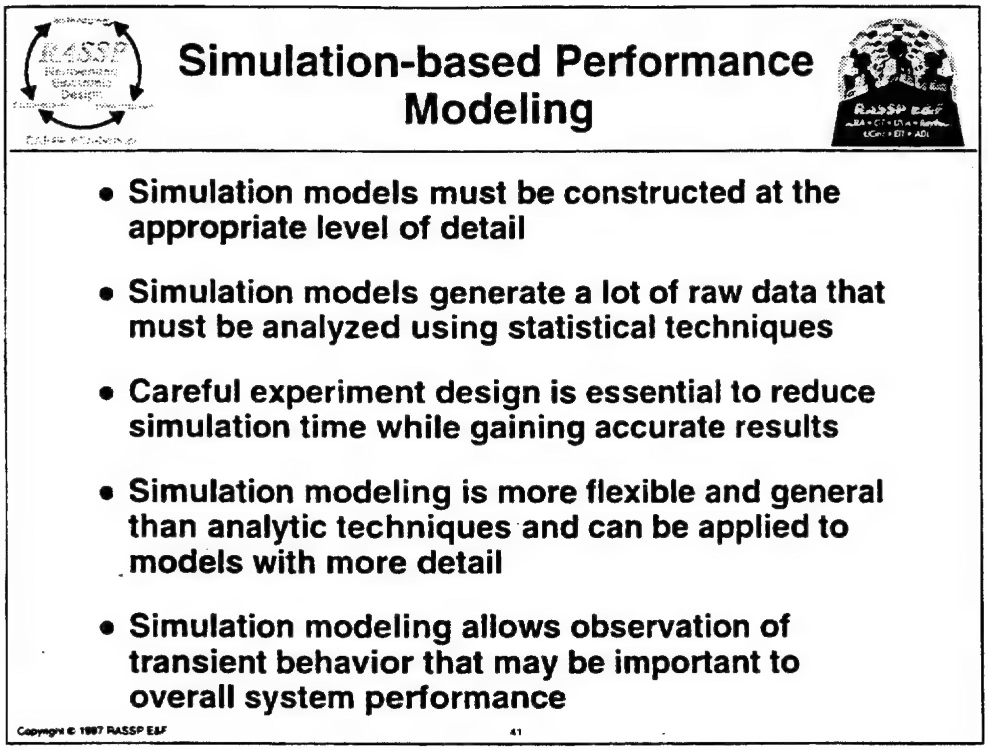


# Analytical Performance Modeling

- Constructing a mathematical model of the system behavior and solving it for the metrics of interest
- Analytic models become intractable unless they are small and at a high level of detail
- However, small analytical models:
  - can usually be solved easily and generate accurate results for the general case
  - generate results that have a better predictive value than those generated by simulation
- In addition, construction of large analytic models can give good insight into the system even if they are too difficult to solve

Analytical performance modeling techniques consist of constructing and solving a mathematical model of the system. Their main advantage is their accuracy and the speed with which they can be solved. Their main disadvantage is the fact that they become intractable for all but the smallest systems.

[Kant92]



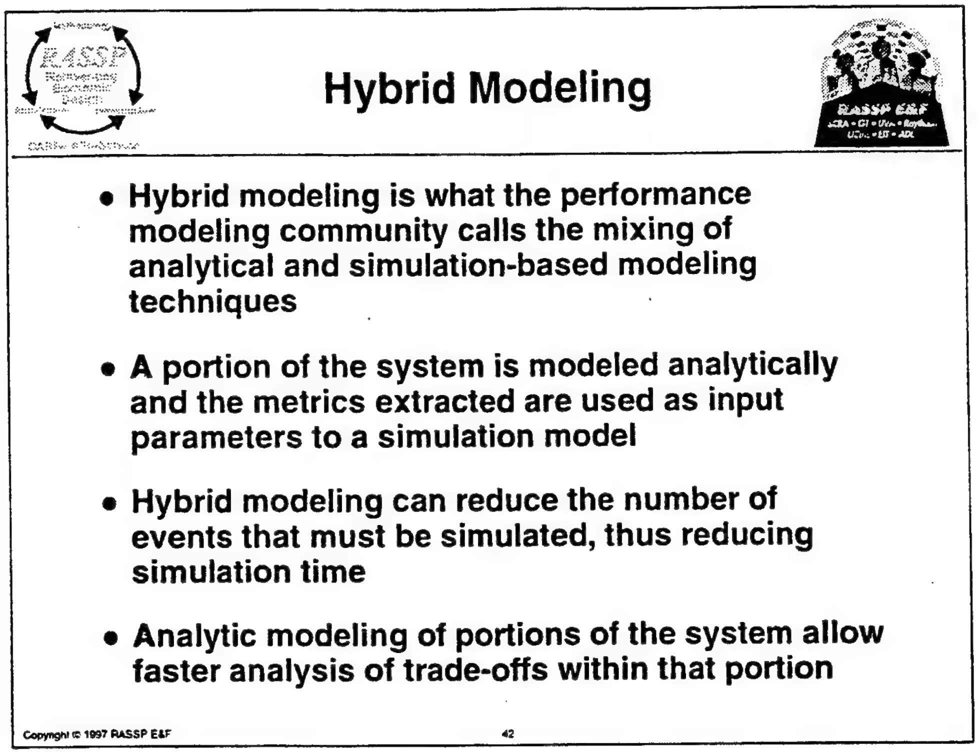
## Simulation-based Performance Modeling

- Simulation models must be constructed at the appropriate level of detail
- Simulation models generate a lot of raw data that must be analyzed using statistical techniques
- Careful experiment design is essential to reduce simulation time while gaining accurate results
- Simulation modeling is more flexible and general than analytic techniques and can be applied to models with more detail
- Simulation modeling allows observation of transient behavior that may be important to overall system performance

Simulation-based techniques consist of constructing and executing a model of the system in a high-level programming language or hardware description language (hdl). Simulation-based models are more generally applicable and can handle larger systems. The simulation execution time can become excessive for very complex systems however, if the level of detail of the model becomes too high. Unlike analytical models, which just give indications of system steady-state behavior, simulation-based models allow observation of the transient behavior of the system which may be important.

In addition, simulation-based models typically generate large amounts of data that have to be analyzed using statistical techniques.

[Kant92]

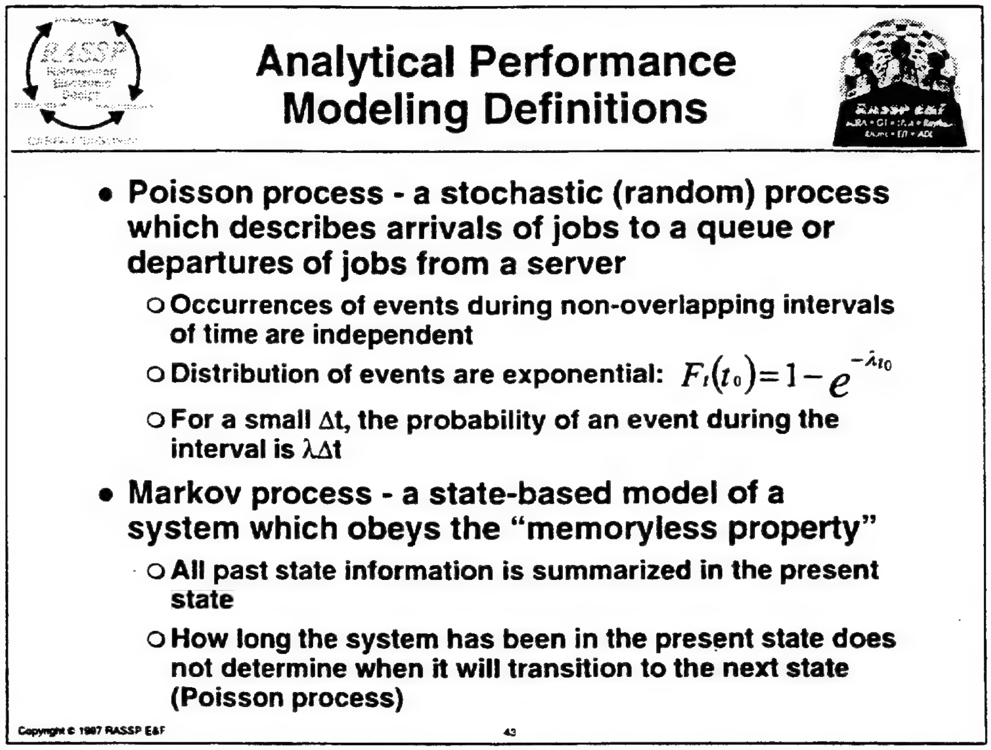


## Hybrid Modeling

- Hybrid modeling is what the performance modeling community calls the mixing of analytical and simulation-based modeling techniques
- A portion of the system is modeled analytically and the metrics extracted are used as input parameters to a simulation model
- Hybrid modeling can reduce the number of events that must be simulated, thus reducing simulation time
- Analytic modeling of portions of the system allow faster analysis of trade-offs within that portion

Hybrid modeling is the term used in the queuing model and Petri Net community to describe mixed analytical and simulation based performance modeling. It is a somewhat overloaded term in that hybrid modeling has also been used to describe the mixture of performance and behavioral models although the preferred term for that is “mixed level modeling.”

Hybrid modeling attempts to incorporate the benefits of both analytical and simulation-based modeling techniques.

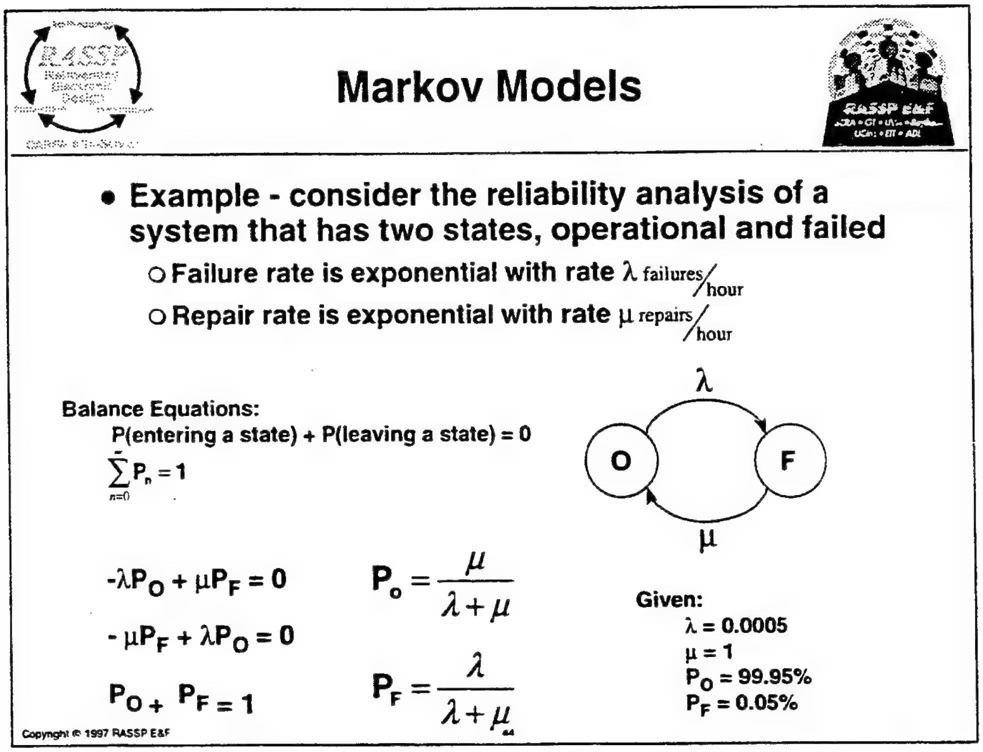


## Analytical Performance Modeling Definitions

- **Poisson process - a stochastic (random) process which describes arrivals of jobs to a queue or departures of jobs from a server**
  - **Occurrences of events during non-overlapping intervals of time are independent**
  - **Distribution of events are exponential:** $F_t(t_0)=1-e^{-\lambda t_0}$
  - **For a small Δt, the probability of an event during the interval is λΔt**
- **Markov process - a state-based model of a system which obeys the "memoryless property"**
  - **All past state information is summarized in the present state**
  - **How long the system has been in the present state does not determine when it will transition to the next state (Poisson process)**

Most analytical performance modeling techniques are based on a Poisson process. This is a stochastic process in which the distribution of events are exponential and occurrences of events in non-overlapping time intervals are independent. Because of this property, the probability that an event occurs in a small interval of time is proportional to the probability distribution.

The Markov model is the basic modeling paradigm. A Markov model is a state based model where the probability of transitioning from one state to another is a Poisson process. This allows the model to be easily solved as will be seen.



## Markov Models

- **Example - consider the reliability analysis of a system that has two states, operational and failed**
  - ○ **Failure rate is exponential with rate** λ failures/hour
  - ○ **Repair rate is exponential with rate** μ repairs/hour

**Balance Equations:**

P(entering a state) + P(leaving a state) = 0

$$\sum_{n=0} P_n = 1$$

$$-\lambda P_O + \mu P_F = 0 \qquad P_o = \frac{\mu}{\lambda+\mu}$$

$$-\mu P_F + \lambda P_O = 0$$

$$P_O + P_F = 1 \qquad P_F = \frac{\lambda}{\lambda+\mu}$$

**Given:**

$\lambda = 0.0005$

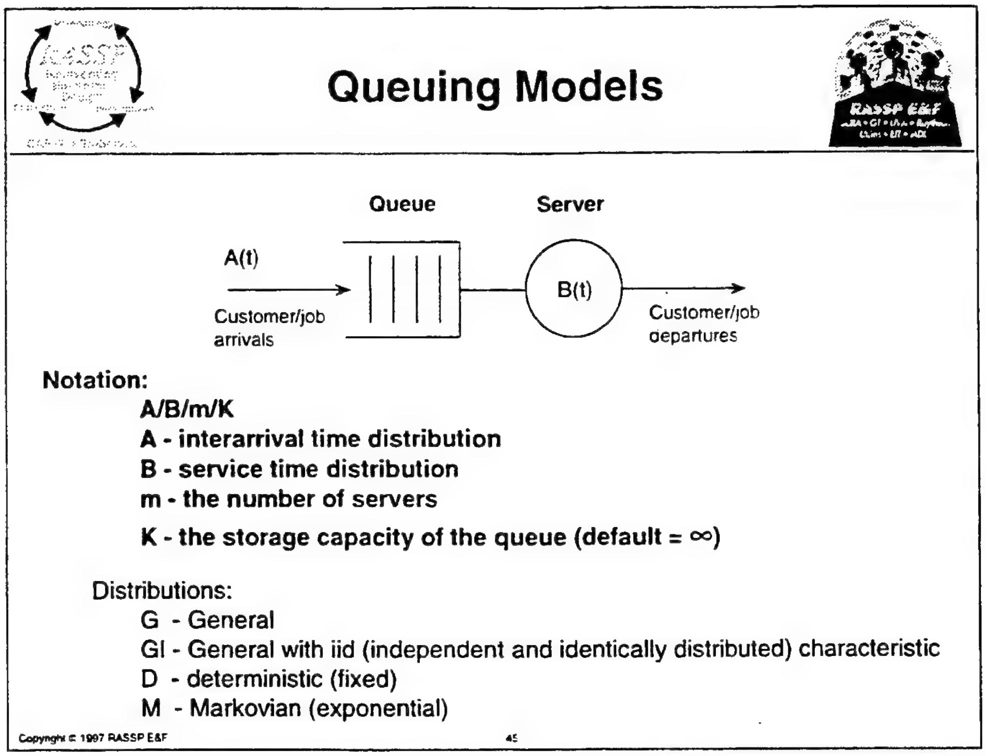
$\mu = 1$

$P_O = 99.95\%$

$P_F = 0.05\%$

This simple two state example (even though it is derived from reliability analysis) shows how a Markov model is solved.

Balance equations that are derived from the fact that the sum of all probabilities entering a state must be equal to all probabilities leaving that state and all probabilities must sum to 1. These balance equations can then be solved to determine the probability of being in each state.



## Queuing Models

Queue
Server
A(t)
Customer/job arrivals
B(t)
Customer/job departures

**Notation:**

**A/B/m/K**
**A - interarrival time distribution**
**B - service time distribution**
**m - the number of servers**
**K - the storage capacity of the queue (default = ∞)**

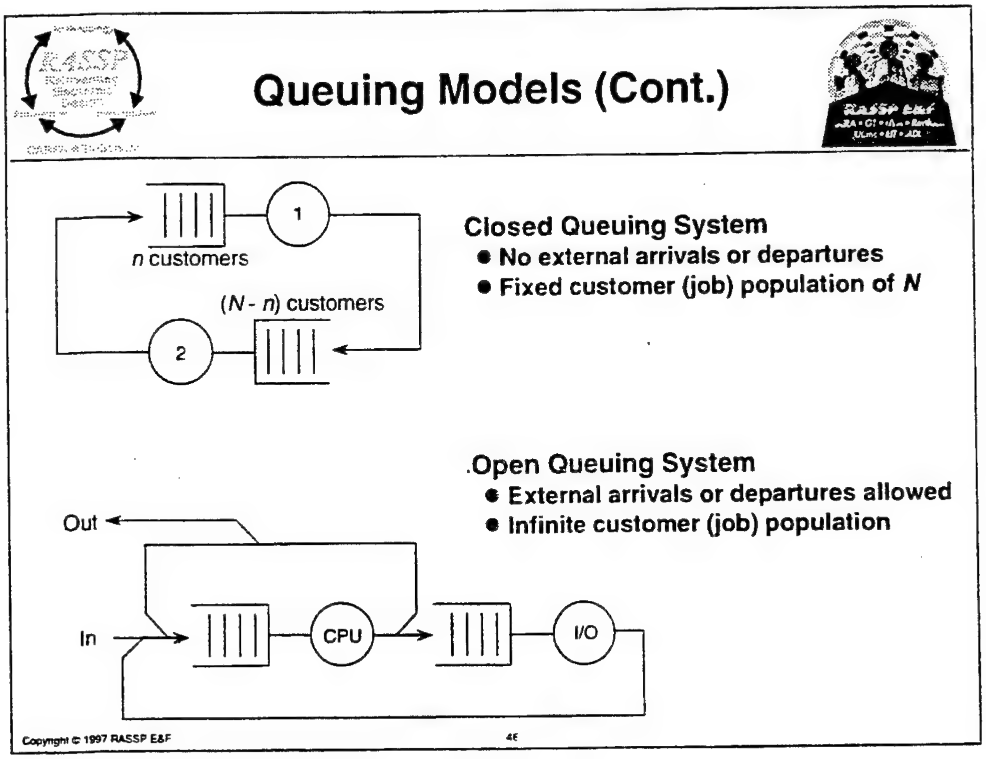
Distributions:
G - General
GI - General with iid (independent and identically distributed) characteristic
D - deterministic (fixed)
M - Markovian (exponential)

This slide describes the convention with which queues are specified.

The discussion here will be limited to M/M queues since they can be described as Markov models, as will be shown.

iid - independent and identically distributed

[Cassandras93] has probably the best description of queuing networks and how they can be analyzed as Markov models, but [Sauer81] is also good and has some good examples.

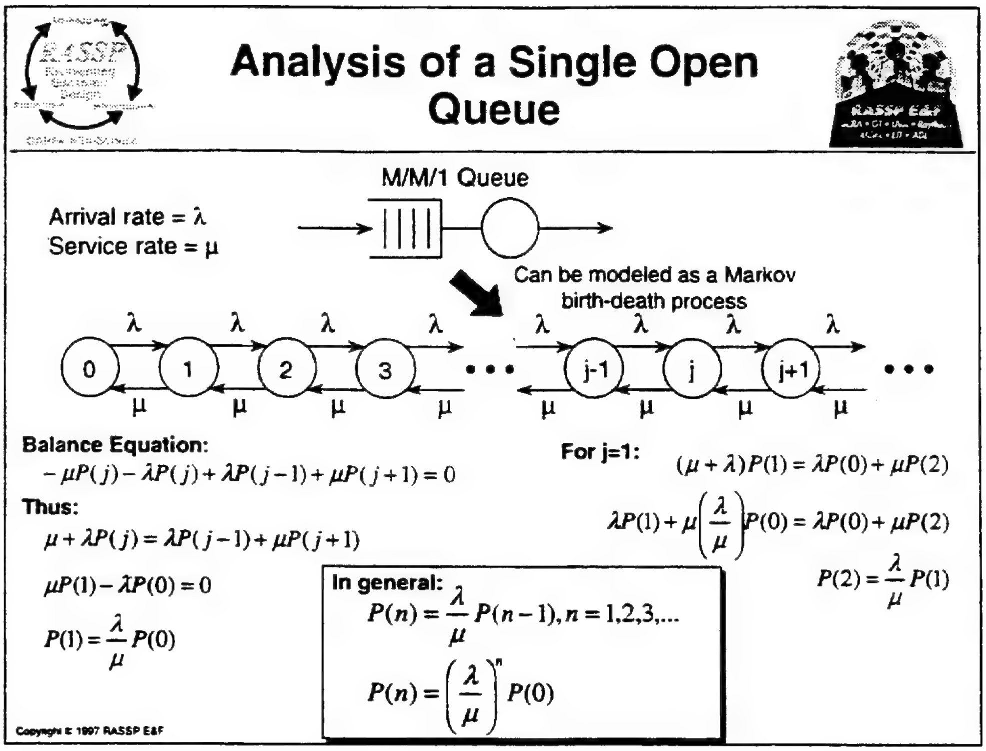


## Queuing Models (Cont.)

**Closed Queuing System**
- No external arrivals or departures
- Fixed customer (job) population of $N$

.**Open Queuing System**
- External arrivals or departures allowed
- Infinite customer (job) population

Both open and closed queuing networks can be analyzed, but there must be some restrictions on the arrivals and departures in an open queuing network so that it may be analyzed using these techniques.



# Analysis of a Single Open Queue

M/M/1 Queue

Arrival rate = λ
Service rate = μ

Can be modeled as a Markov birth-death process

**Balance Equation:**

$$-\mu P(j) - \lambda P(j) + \lambda P(j-1) + \mu P(j+1) = 0$$

**Thus:**

$$\mu + \lambda P(j) = \lambda P(j-1) + \mu P(j+1)$$

$$\mu P(1) - \lambda P(0) = 0$$

$$P(1) = \frac{\lambda}{\mu} P(0)$$

**In general:**

$$P(n) = \frac{\lambda}{\mu} P(n-1), n = 1,2,3,\ldots$$

$$P(n) = \left(\frac{\lambda}{\mu}\right)^n P(0)$$

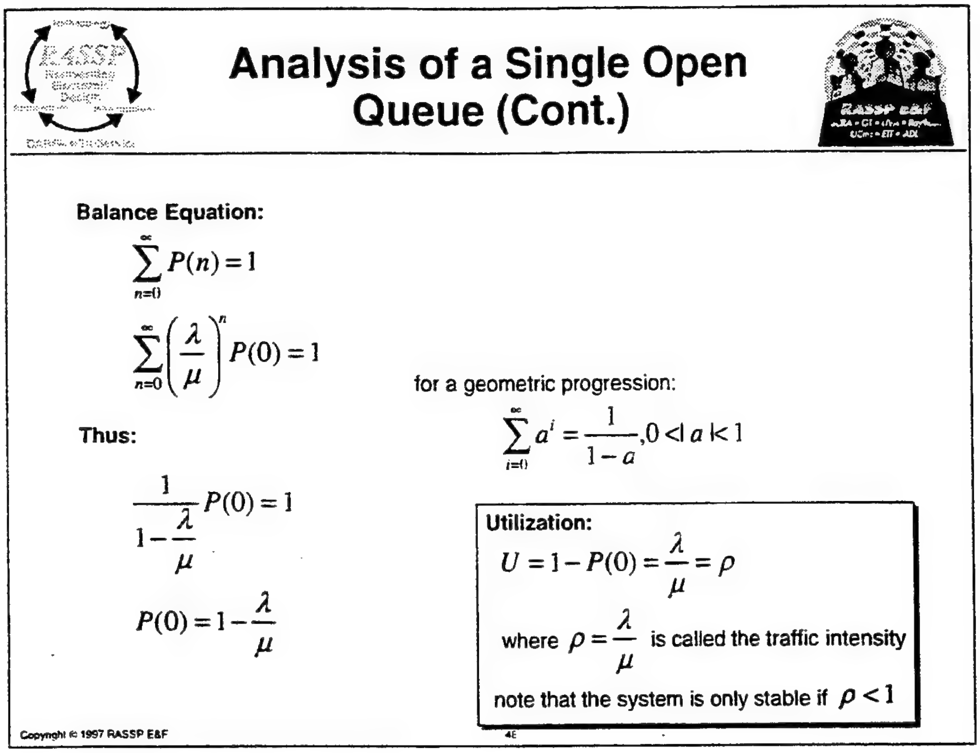
**For j=1:**

$$(\mu + \lambda) P(1) = \lambda P(0) + \mu P(2)$$

$$\lambda P(1) + \mu\left(\frac{\lambda}{\mu}\right) P(0) = \lambda P(0) + \mu P(2)$$

$$P(2) = \frac{\lambda}{\mu} P(1)$$

This slides shows the analysis if a single queue if infinite size with exponential arrival and service rates. As shown, the queue can be modeled with a Markov birth-death process. This allows the steady state behavior of the queue to be modeled analytically. Note that the service rate must be greater than the arrival rate for the model to be stable.



## Analysis of a Single Open Queue (Cont.)

**Balance Equation:**

$$\sum_{n=0}^{\infty} P(n) = 1$$

$$\sum_{n=0}^{\infty} \left(\frac{\lambda}{\mu}\right)^n P(0) = 1$$

for a geometric progression:

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}, 0 < | a | < 1$$

**Thus:**

$$\frac{1}{1-\frac{\lambda}{\mu}} P(0) = 1$$

$$P(0) = 1 - \frac{\lambda}{\mu}$$

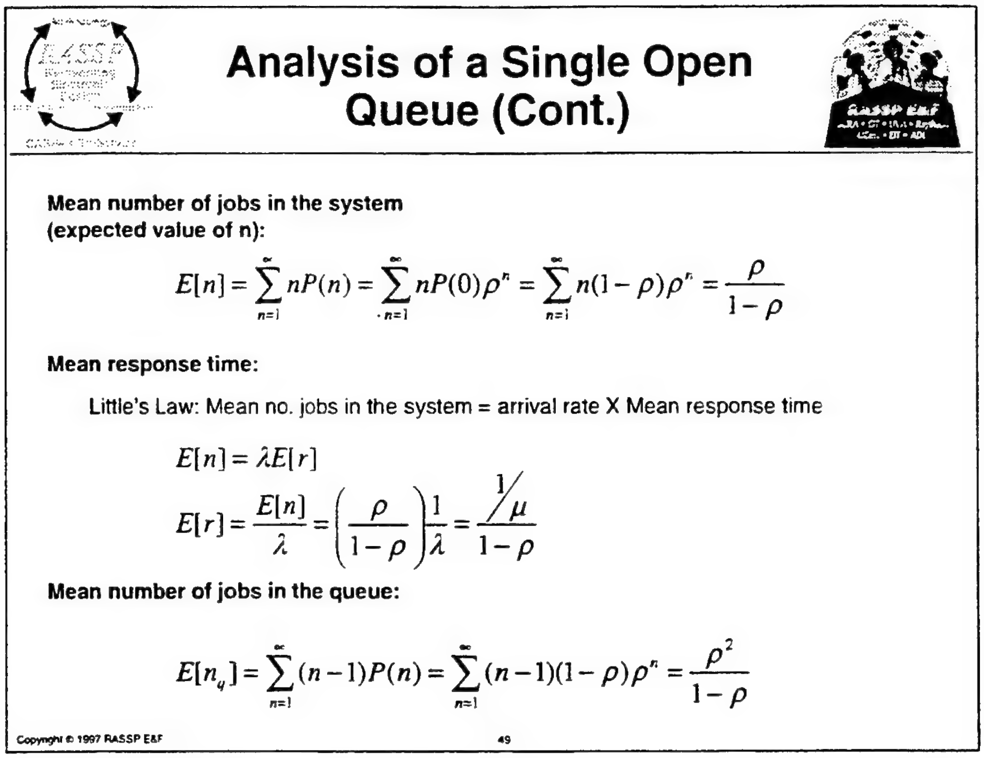
**Utilization:**

$$U = 1 - P(0) = \frac{\lambda}{\mu} = \rho$$

where $\rho = \frac{\lambda}{\mu}$ is called the traffic intensity

note that the system is only stable if $\rho < 1$

This is calculation of utilization of the server in the single queue system.



## Analysis of a Single Open Queue (Cont.)

**Mean number of jobs in the system (expected value of n):**

$$E[n] = \sum_{n=1}^{\infty} nP(n) = \sum_{n=1}^{\infty} nP(0)\rho^n = \sum_{n=1}^{\infty} n(1-\rho)\rho^n = \frac{\rho}{1-\rho}$$

**Mean response time:**

Little's Law: Mean no. jobs in the system = arrival rate X Mean response time

$$E[n] = \lambda E[r]$$

$$E[r] = \frac{E[n]}{\lambda} = \left(\frac{\rho}{1-\rho}\right)\frac{1}{\lambda} = \frac{1/\mu}{1-\rho}$$

**Mean number of jobs in the queue:**

$$E[n_q] = \sum_{n=1}^{\infty} (n-1)P(n) = \sum_{n=1}^{\infty} (n-1)(1-\rho)\rho^n = \frac{\rho^2}{1-\rho}$$

This is the calculation of mean number of jobs in the system, mean response time, and mean number of jobs in the queue. Note that this slide introduces Little's Law, an important theorem in queue analysis.

193

## Single Queue Analysis Example

- **Consider a network router modeled as an M/M/1 queue:**
  - ○ **Arrival rate λ = 1000 packets per second**
  - ○ **Routing takes an average of 150 µs** µ = 1/150 µs = 6666 pps

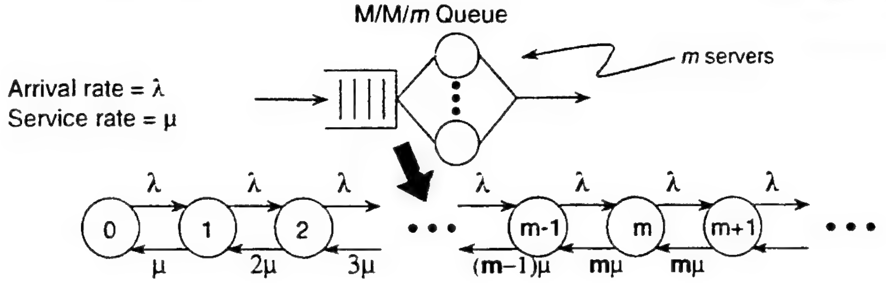
Router utilization: $U = \rho = \frac{\lambda}{\mu} = \frac{1000}{6666} = 15\%$

Mean number of packets in the router: $E[n] = \frac{\rho}{1-\rho} = \frac{1000/6666}{1 - 1000/6666} = 0.176$

Mean time spent in the router: $E[r] = \frac{1/\mu}{1-\rho} = \frac{1/6666}{1 - 1000/6666} = 176.5\ \mu s$

This is an example of how a real life system can be analyzed as a M/M/1 queue. Note that the analysis of a system with a limited queue size (M/M/1/N), which covers more real-life systems, is equally simple.

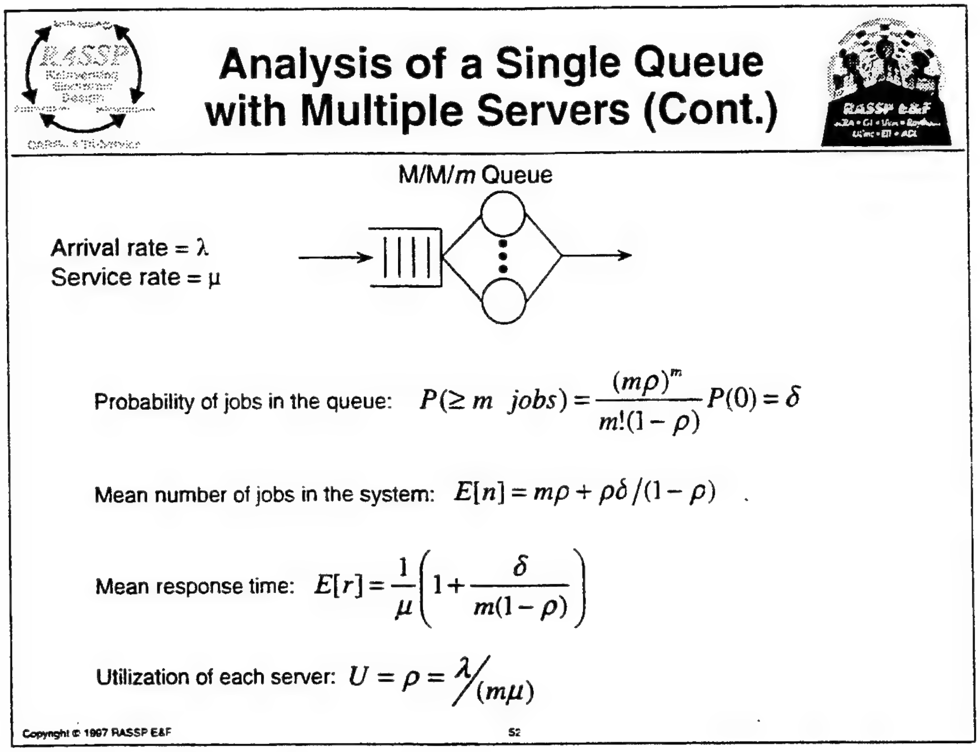
# Analysis of a Single Queue with Multiple Servers



Probability of zero jobs in the system: $P(0) = \left[ 1 + \frac{(m\rho)^m}{m!(1-\rho)} + \sum_{n=1}^{m-1} \frac{(m\rho)^n}{n!} \right]^{-1}$

Probability of $n$ jobs in the system: $P(n) = \begin{cases} P(0)\frac{(m\rho)^n}{n!}, n < m \\ P(0)\frac{\rho^n m^m}{m!}, n \geq m \end{cases}$

This is the analysis if an M/M/n system, one with a single exponential queue but multiple servers, e.g. a multiprocessor system for transaction processing.



## Analysis of a Single Queue with Multiple Servers (Cont.)

M/M/*m* Queue

Arrival rate = λ
Service rate = μ

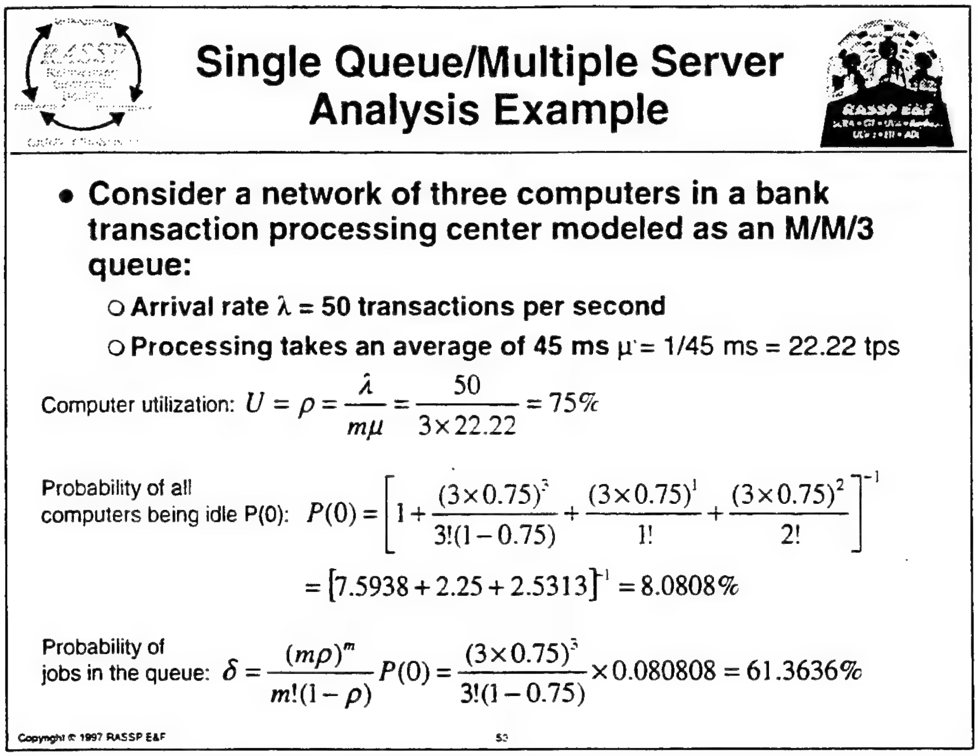
Probability of jobs in the queue: $P(\geq m \ \ jobs) = \frac{(m\rho)^m}{m!(1-\rho)} P(0) = \delta$

Mean number of jobs in the system: $E[n] = m\rho + \rho\delta/(1-\rho)$

Mean response time: $E[r] = \frac{1}{\mu}\left(1 + \frac{\delta}{m(1-\rho)}\right)$

Utilization of each server: $U = \rho = {}^{\lambda}\!/_{(m\mu)}$

This is the remainder of the analysis.



## Single Queue/Multiple Server Analysis Example

- **Consider a network of three computers in a bank transaction processing center modeled as an M/M/3 queue:**
  - ○ **Arrival rate λ = 50 transactions per second**
  - ○ **Processing takes an average of 45 ms** $\mu$ = 1/45 ms = 22.22 tps

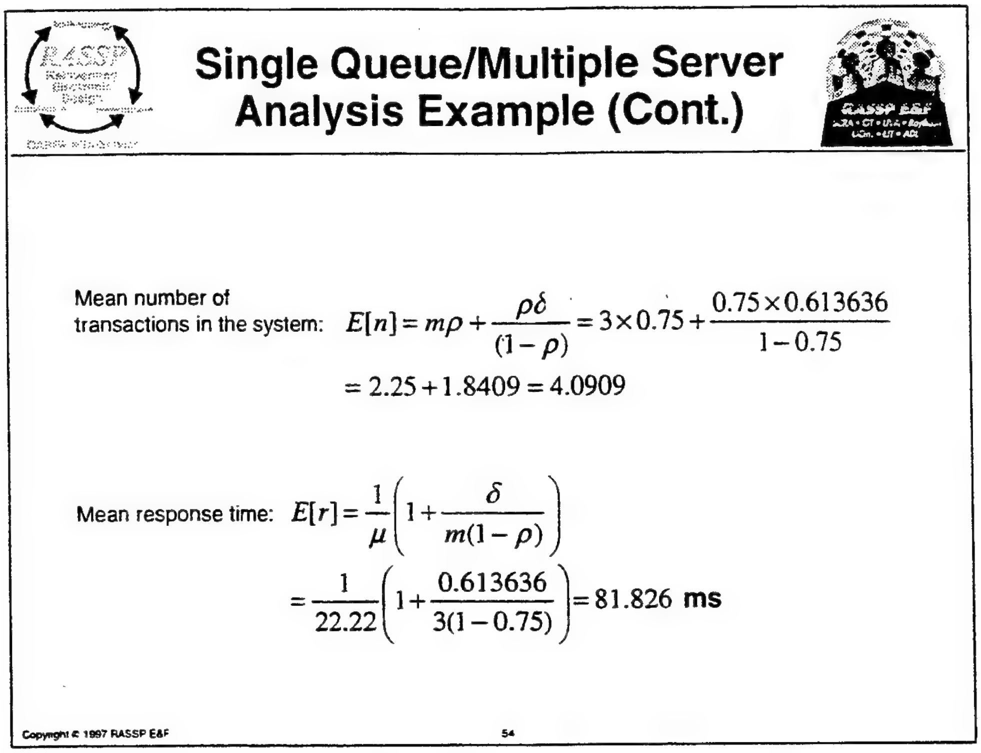
Computer utilization: $U = \rho = \frac{\lambda}{m\mu} = \frac{50}{3\times 22.22} = 75\%$

Probability of all computers being idle P(0): $P(0) = \left[1 + \frac{(3\times 0.75)^3}{3!(1-0.75)} + \frac{(3\times 0.75)^1}{1!} + \frac{(3\times 0.75)^2}{2!}\right]^{-1}$

$$= [7.5938 + 2.25 + 2.5313]^{-1} = 8.0808\%$$

Probability of jobs in the queue: $\delta = \frac{(m\rho)^m}{m!(1-\rho)}P(0) = \frac{(3\times 0.75)^3}{3!(1-0.75)}\times 0.080808 = 61.3636\%$

An example of an M/M/n queue model.



## Single Queue/Multiple Server Analysis Example (Cont.)

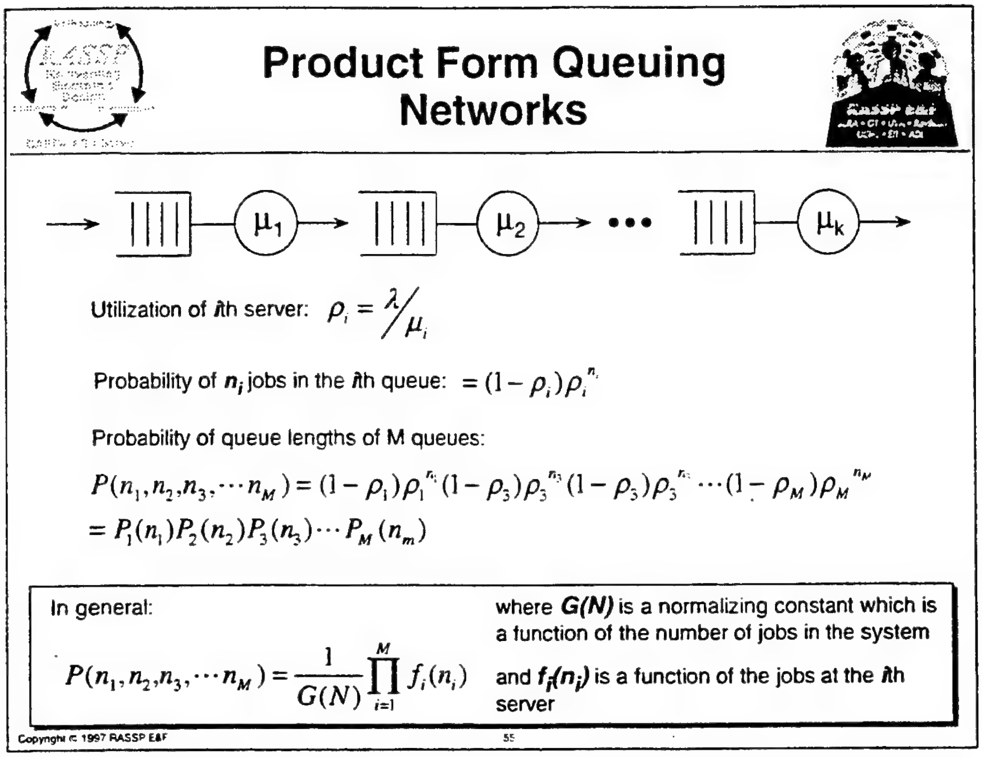
Mean number of transactions in the system: $E[n] = m\rho + \frac{\rho\delta}{(1-\rho)} = 3 \times 0.75 + \frac{0.75 \times 0.613636}{1-0.75}$

$$= 2.25 + 1.8409 = 4.0909$$

Mean response time: $E[r] = \frac{1}{\mu}\left(1 + \frac{\delta}{m(1-\rho)}\right)$

$$= \frac{1}{22.22}\left(1 + \frac{0.613636}{3(1-0.75)}\right) = 81.826 \text{ ms}$$

M/M/n example continued.



## Product Form Queuing Networks

Utilization of *i*th server: $\rho_i = \lambda / \mu_i$

Probability of $n_i$ jobs in the *i*th queue: $= (1-\rho_i)\rho_i^{n_i}$

Probability of queue lengths of M queues:

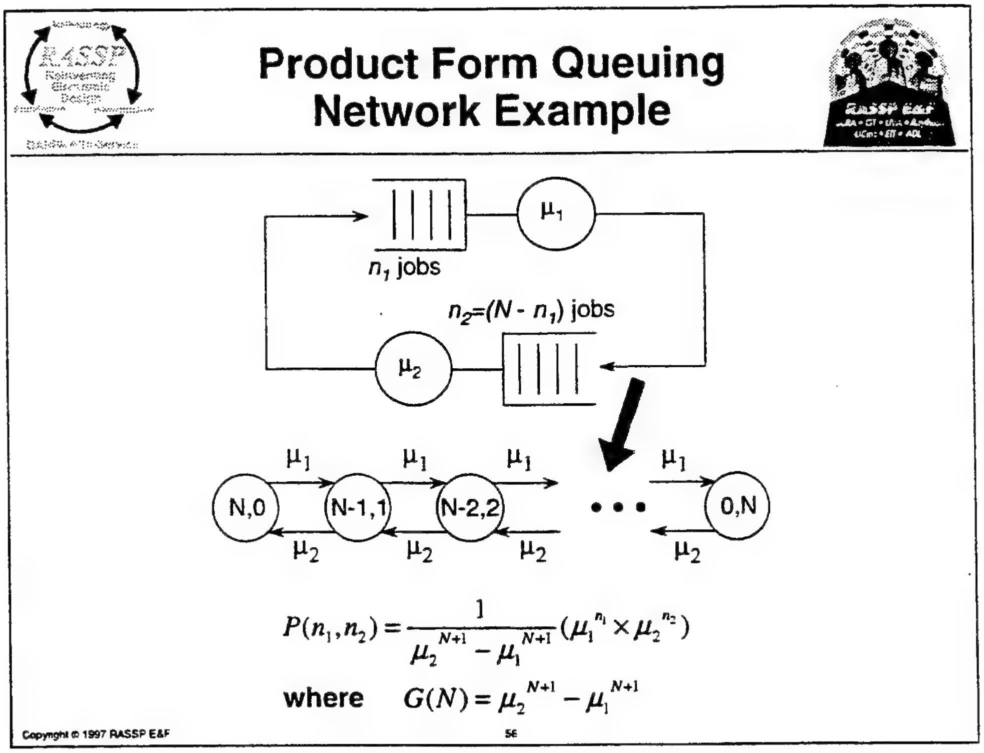
$$P(n_1,n_2,n_3,\cdots n_M) = (1-\rho_1)\rho_1^{n_1}(1-\rho_3)\rho_3^{n_3}(1-\rho_3)\rho_3^{n_3}\cdots(1-\rho_M)\rho_M^{n_M}$$
$$= P_1(n_1)P_2(n_2)P_3(n_3)\cdots P_M(n_m)$$

In general:

$$P(n_1,n_2,n_3,\cdots n_M) = \frac{1}{G(N)}\prod_{i=1}^{M} f_i(n_i)$$

where ***G(N)*** is a normalizing constant which is a function of the number of jobs in the system and $f_i(n_i)$ is a function of the jobs at the *i*th server

This is a brief presentation of the analysis of a chain of M/M/1 queues. Note the form that the solution takes is the general form of the solution of a closed network of M/M/1 queues. Queuing networks whose solution takes this form are called “product form networks.”

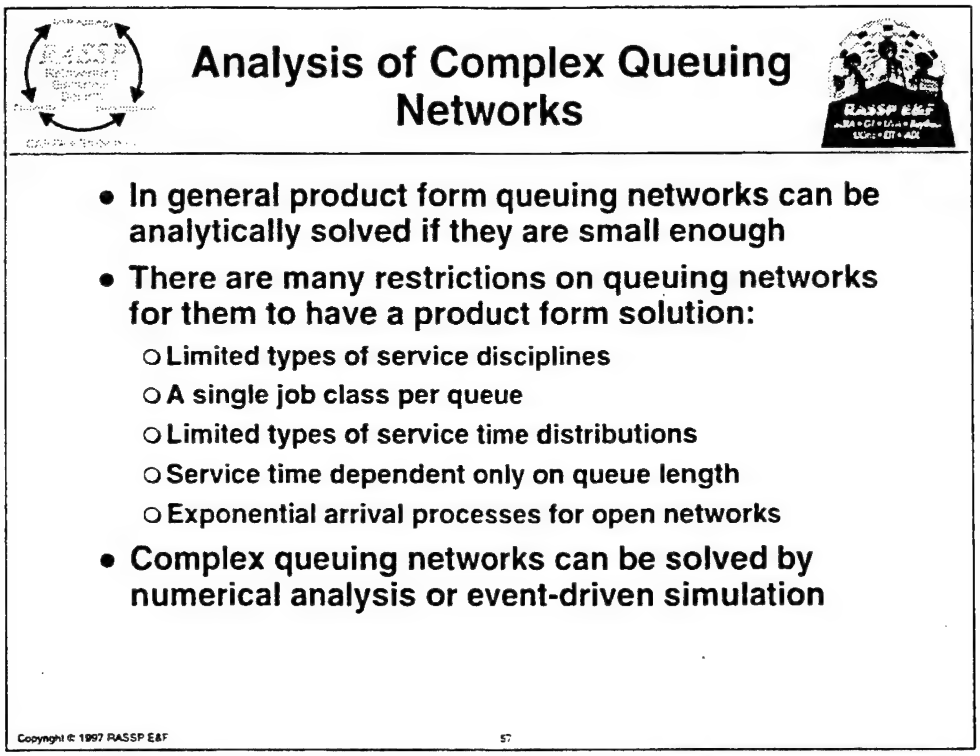


## Product Form Queuing Network Example

$$P(n_1,n_2)=\frac{1}{\mu_2^{N+1}-\mu_1^{N+1}}(\mu_1^{n_1}\times\mu_2^{n_2})$$

**where** $G(N)=\mu_2^{N+1}-\mu_1^{N+1}$

This is an example of the solution of a closed network of M/M/1 queues. Note how the solution takes the general product form.

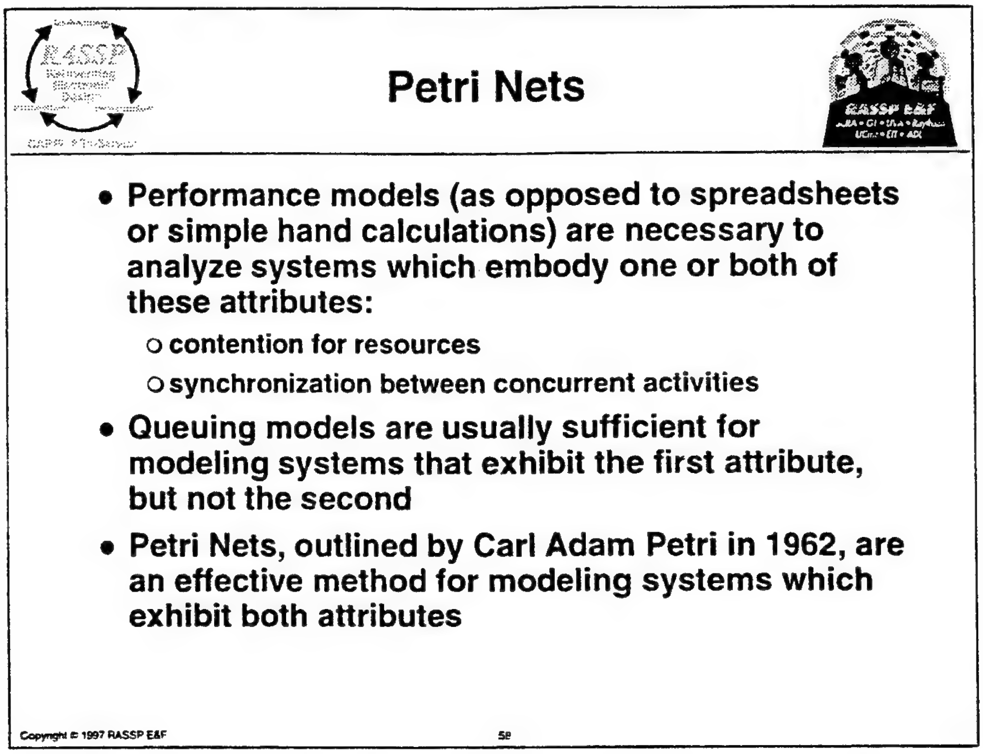


## Analysis of Complex Queuing Networks

- In general product form queuing networks can be analytically solved if they are small enough
- There are many restrictions on queuing networks for them to have a product form solution:
  - ○ Limited types of service disciplines
  - ○ A single job class per queue
  - ○ Limited types of service time distributions
  - ○ Service time dependent only on queue length
  - ○ Exponential arrival processes for open networks
- Complex queuing networks can be solved by numerical analysis or event-driven simulation

Product form queuing networks have a very mathematically "clean" solution, but there are many restrictions on the queuing networks such that they are "product form networks."

Note that complex queuing networks can be solved numerically or by event driven simulation. This is the basis of many performance tools like SES Workbench, Extend, Foresight, etc.



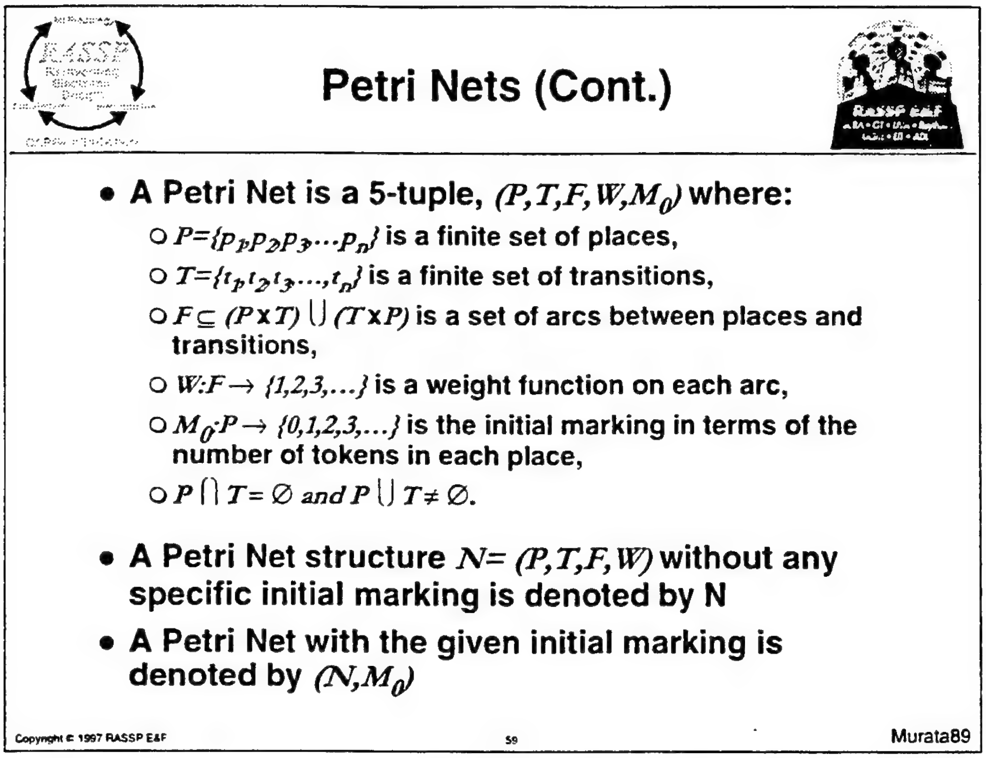
## Petri Nets

- **Performance models (as opposed to spreadsheets or simple hand calculations) are necessary to analyze systems which embody one or both of these attributes:**
  - ○ **contention for resources**
  - ○ **synchronization between concurrent activities**
- **Queuing models are usually sufficient for modeling systems that exhibit the first attribute, but not the second**
- **Petri Nets, outlined by Carl Adam Petri in 1962, are an effective method for modeling systems which exhibit both attributes**

For simple systems that do not exhibit concurrency and contention, detailed performance modeling may not be necessary, a simple "spread sheet" approach might suffice. For systems that exhibit concurrency, and contention (like the transaction system example), queuing models are applicable. However, for systems that exhibit synchronization between concurrent activities, queuing models are not adequate.

Petri Nets, developed in 1962, are suited to modeling systems that have concurrency, contention, and synchronization.

The major reference for Petri Nets is the paper by Murata [Murata89], but [Cassandras93] is a good text reference.

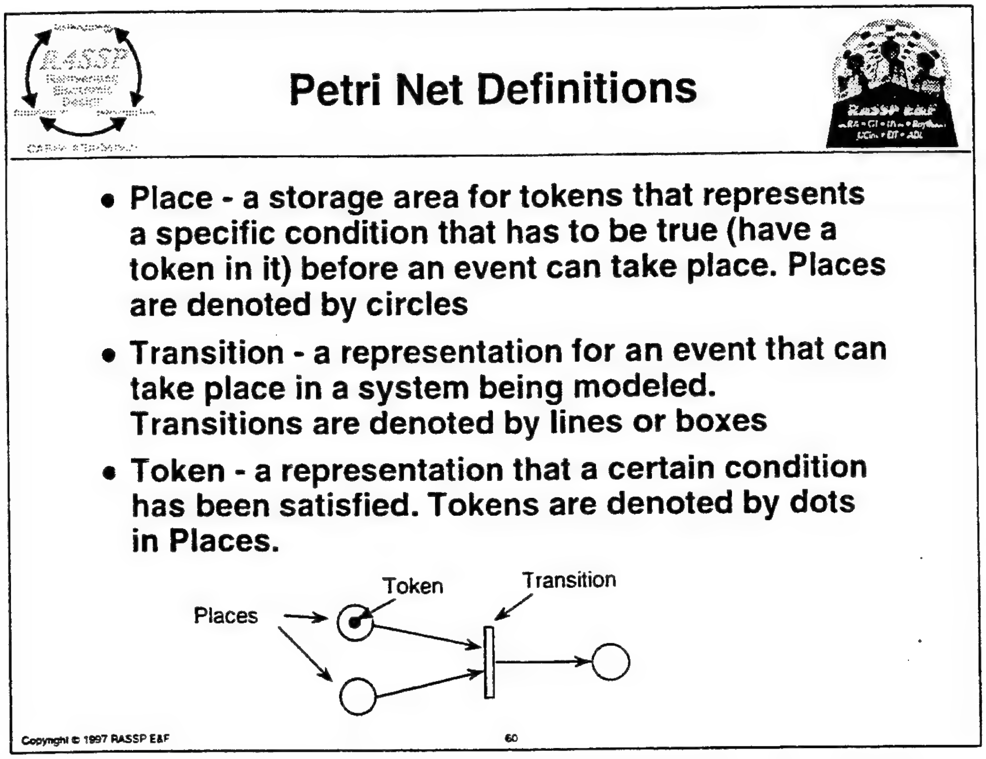


## Petri Nets (Cont.)

- A Petri Net is a 5-tuple, $(P,T,F,W,M_0)$ where:
  - ○ $P=\{p_1,p_2,p_3,\ldots p_n\}$ is a finite set of places,
  - ○ $T=\{t_1,t_2,t_3,\ldots,t_n\}$ is a finite set of transitions,
  - ○ $F\subseteq (P\times T)\bigcup(T\times P)$ is a set of arcs between places and transitions,
  - ○ $W{:}F\rightarrow \{1,2,3,\ldots\}$ is a weight function on each arc,
  - ○ $M_0{:}P\rightarrow \{0,1,2,3,\ldots\}$ is the initial marking in terms of the number of tokens in each place,
  - ○ $P\bigcap T=\varnothing$ and $P\bigcup T\neq\varnothing$.
- A Petri Net structure $N=(P,T,F,W)$ without any specific initial marking is denoted by N
- A Petri Net with the given initial marking is denoted by $(N,M_0)$

Murata89

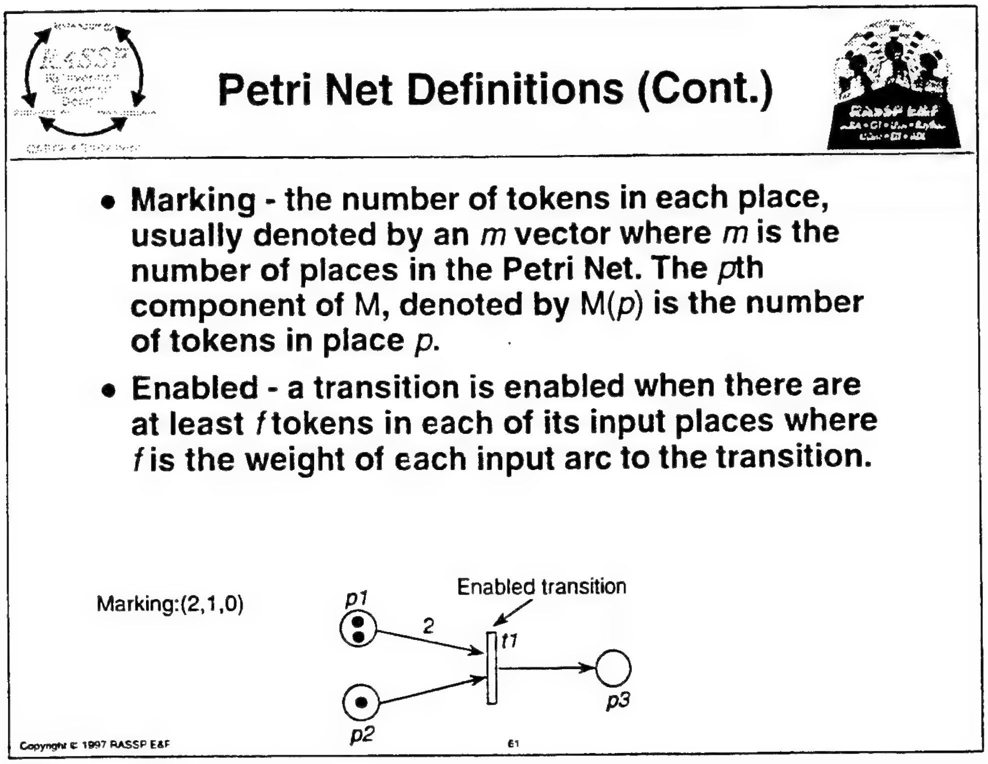
This is the basic definition of a Petri Net. Note that the basic Petri Net contains no notion of time or values on the data modeled in the system.



## Petri Net Definitions

- **Place - a storage area for tokens that represents a specific condition that has to be true (have a token in it) before an event can take place. Places are denoted by circles**
- **Transition - a representation for an event that can take place in a system being modeled. Transitions are denoted by lines or boxes**
- **Token - a representation that a certain condition has been satisfied. Tokens are denoted by dots in Places.**

The basic definitions of the things that make up a Petri Net. Note that the Petri Net definition of a token is slightly different than the definition that will be used in the uninterpreted modeling section. In a Petri Net, a token is a representation that a certain condition, that will cause a transition to fire, has been satisfied. It does not necessarily denote actual data that is moving in the system, as is the case with most (but not all) uninterpreted modeling systems.

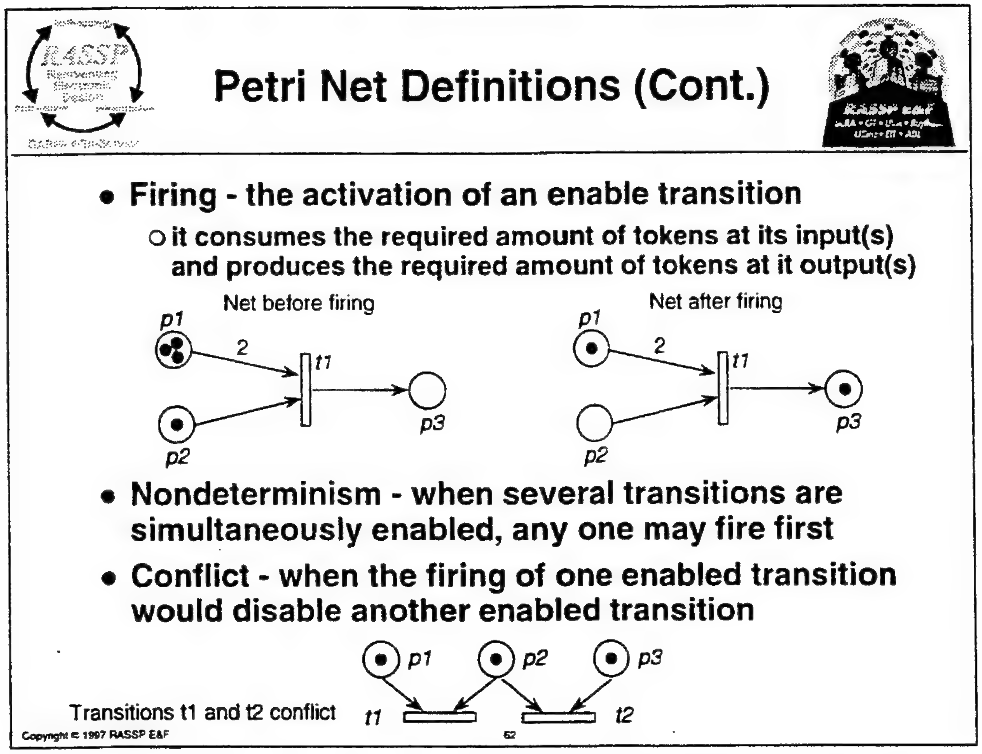


## Petri Net Definitions (Cont.)

- **Marking - the number of tokens in each place, usually denoted by an *m* vector where *m* is the number of places in the Petri Net. The *p*th component of M, denoted by M(*p*) is the number of tokens in place *p*.**
- **Enabled - a transition is enabled when there are at least *f* tokens in each of its input places where *f* is the weight of each input arc to the transition.**

Marking:(2,1,0)
p1
2
Enabled transition
t1
p3
p2

No additional notes necessary.



## Petri Net Definitions (Cont.)

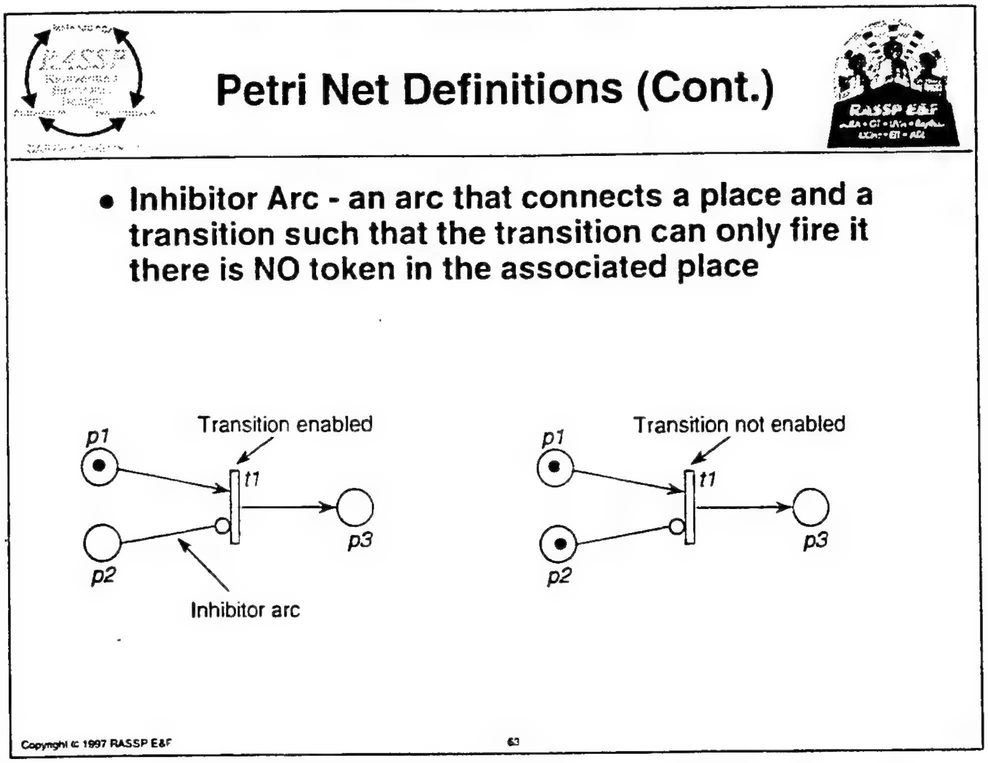
- **Firing - the activation of an enable transition**
  - **it consumes the required amount of tokens at its input(s) and produces the required amount of tokens at it output(s)**

Net before firing
p1
2
t1
p3
p2
Net after firing
p1
2
t1
p3
p2

- **Nondeterminism - when several transitions are simultaneously enabled, any one may fire first**
- **Conflict - when the firing of one enabled transition would disable another enabled transition**

Transitions t1 and t2 conflict

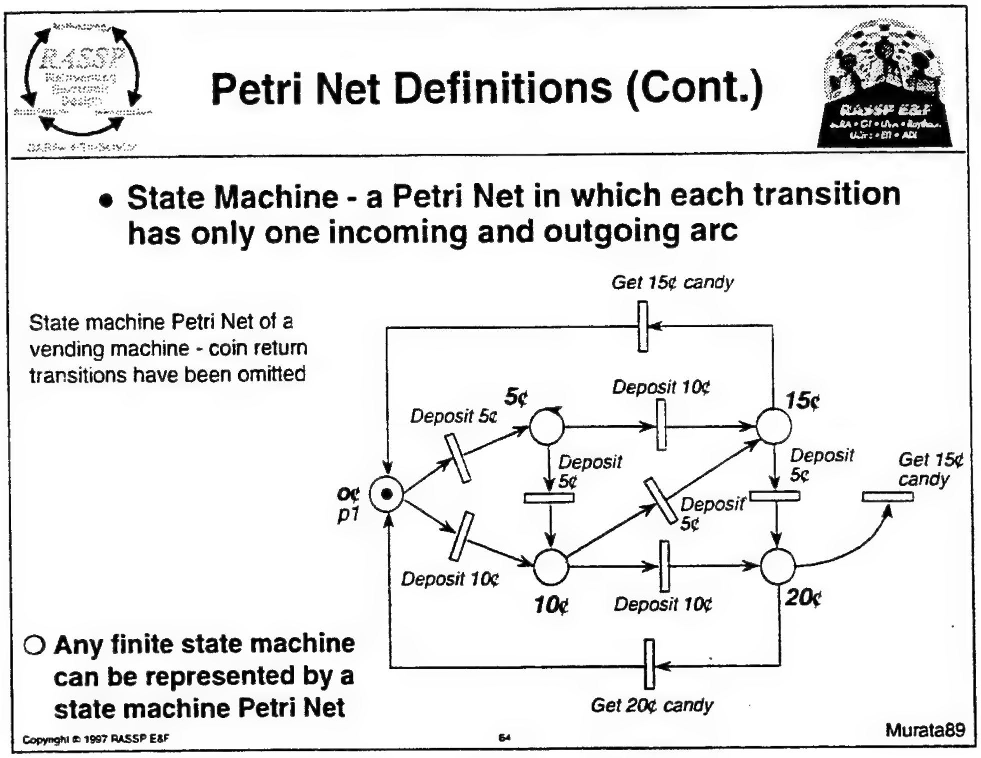
The nondeterminism of Petri Nets is a significant difference between them and other uninterpreted modeling techniques. Where two conflicting transitions are enabled, which one fires first can make a significant difference in how the model behaves.



## Petri Net Definitions (Cont.)

- **Inhibitor Arc - an arc that connects a place and a transition such that the transition can only fire it there is NO token in the associated place**

No additional notes necessary.



## Petri Net Definitions (Cont.)

- **State Machine - a Petri Net in which each transition has only one incoming and outgoing arc**

State machine Petri Net of a vending machine - coin return transitions have been omitted

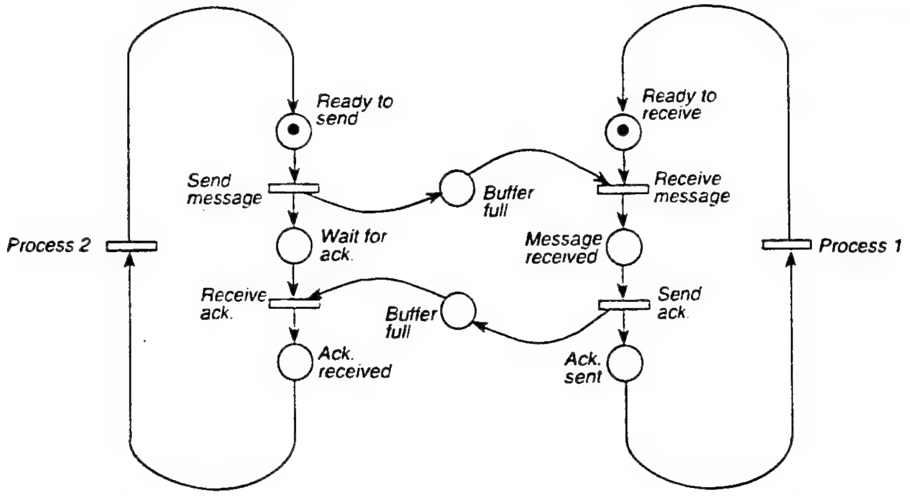
○ **Any finite state machine can be represented by a state machine Petri Net**

64

Murata89

This is a Petri Net model of a finite state machine (FSM). By definition, any FSM can be modeled with a Petri Net. One thing to note here is that in the real state machine, the firing of each transition is triggered by an external event, either the insertion of a coin or the pressing of a "get candy" button. However, in the true Petri Net model, which transition would fire, in the case where two or more are enabled (0¢, 15¢, 20¢ state), is non-deterministic.

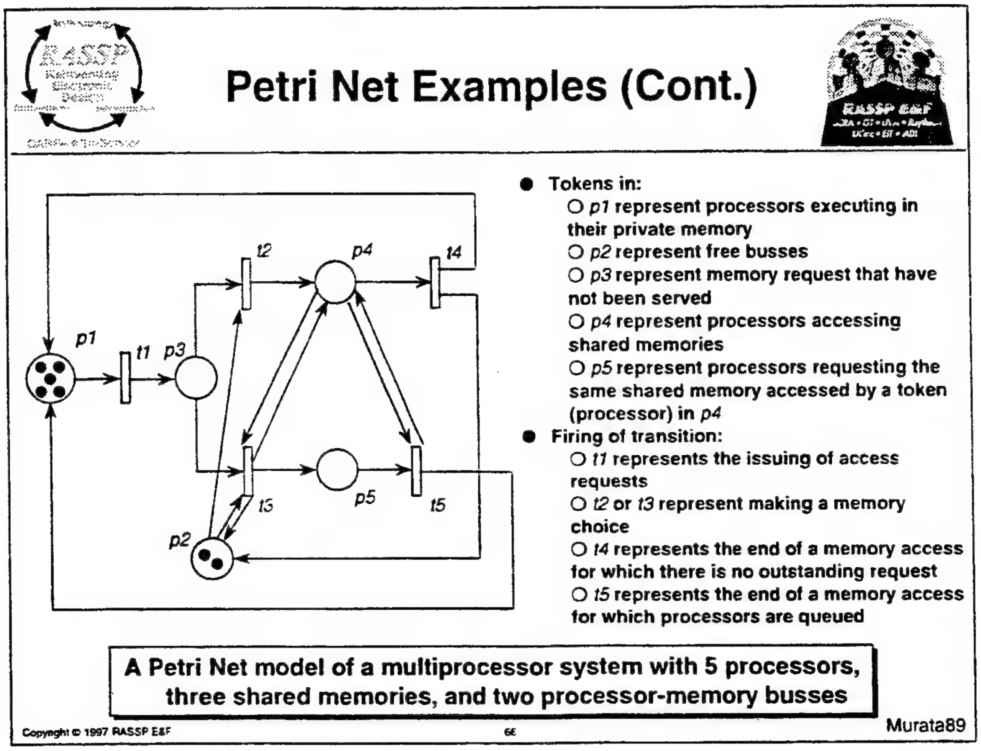
# Petri Net Examples



A Petri Net model of a simple communications protocol

Murata89

This is a Petri Net model of a simple interlocking communications protocol. In fact, both hardware and software systems can be modeled with Petri Nets - a powerful feature.



## Petri Net Examples (Cont.)

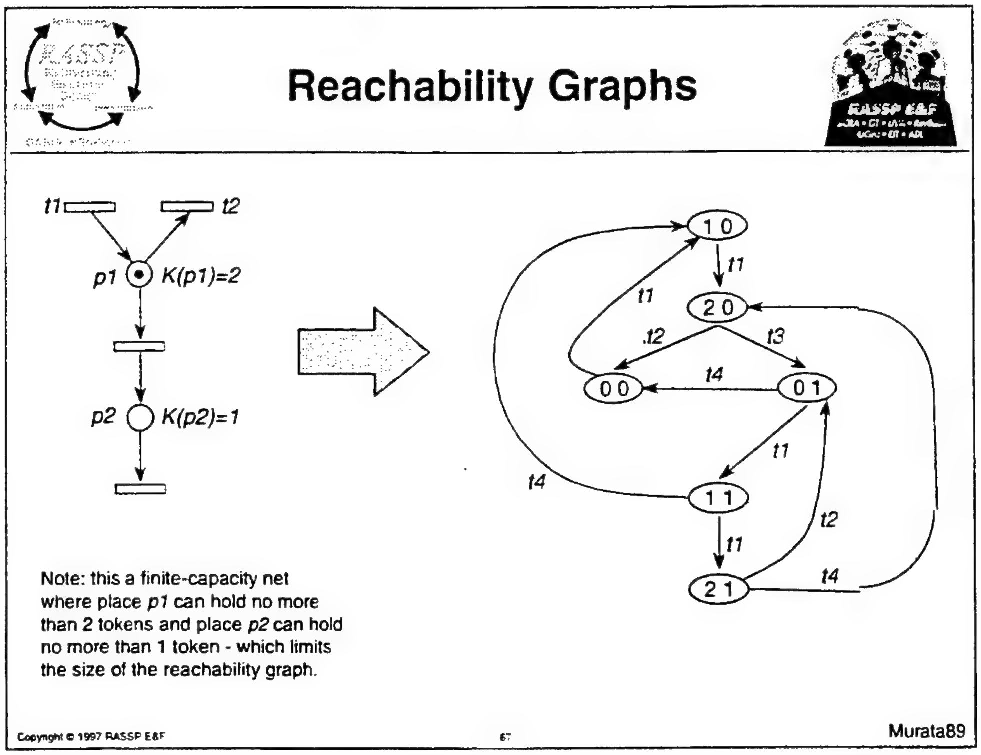
- Tokens in:
  - ○ *p1* represent processors executing in their private memory
  - ○ *p2* represent free busses
  - ○ *p3* represent memory request that have not been served
  - ○ *p4* represent processors accessing shared memories
  - ○ *p5* represent processors requesting the same shared memory accessed by a token (processor) in *p4*
- Firing of transition:
  - ○ *t1* represents the issuing of access requests
  - ○ *t2* or *t3* represent making a memory choice
  - ○ *t4* represents the end of a memory access for which there is no outstanding request
  - ○ *t5* represents the end of a memory access for which processors are queued

**A Petri Net model of a multiprocessor system with 5 processors, three shared memories, and two processor-memory busses**

Murata89

This is a more complex Petri Net model of a multiprocessor system with 5 processors, three shared memories, and two processor-memory busses. It is intended to show how systems of this type can be modeled with Petri Nets and that there is not a one-to-one correspondence between tokens, place, and transitions and hardware components or data packets in a real system - which sometimes makes them difficult to conceive.

See [Murata89] for more details on this example.



## Reachability Graphs

Note: this a finite-capacity net where place *p1* can hold no more than 2 tokens and place *p2* can hold no more than 1 token - which limits the size of the reachability graph.

67

Murata89

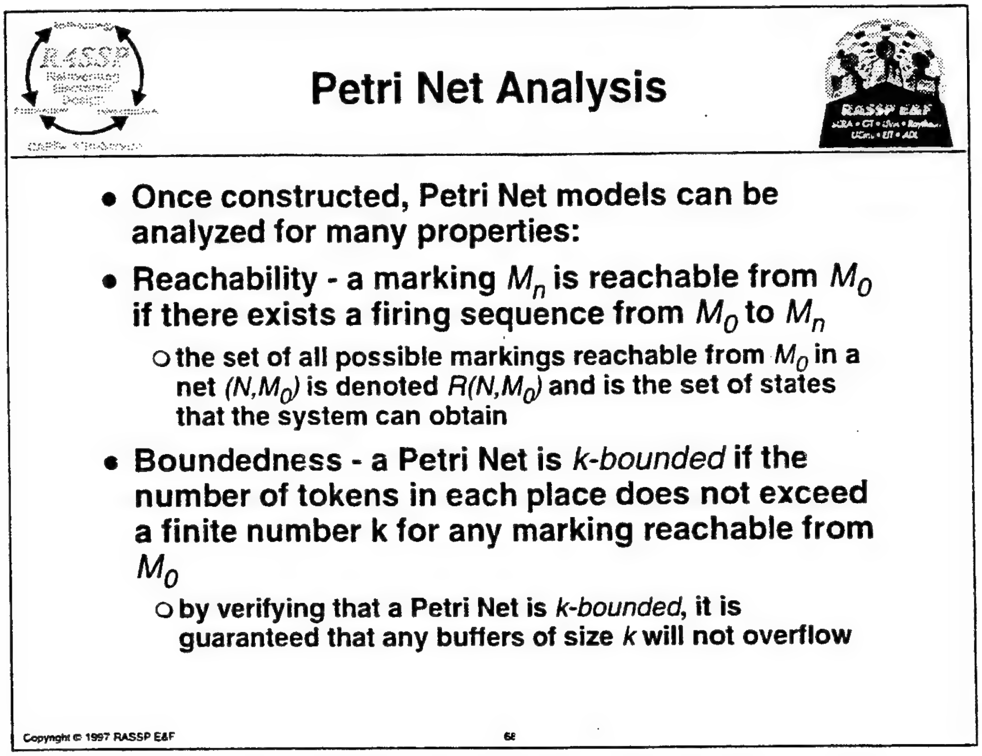
This slide introduces reachability graphs which are representations of the "states" or markings of a Petri Net and how they are reached by various transition firings.

The nodes in the reachability graph are markings (e.g., 1 0 is the marking where there is one token in $p_1$ and 0 tokens in $p_2$.

The arcs in the reachability graph are the transitions that move the Petri Net from one marking to another.

Note that in order to make the reachability graph for this example tractable (as far as drawing it), the example is a finite capacity net in that $p_1$ can hold no more than 2 tokens and $p_2$ can hold no more than 1 token.

Once the reachability graph is constructed, it can be analyzed using various graph algorithms.



# Petri Net Analysis

- **Once constructed, Petri Net models can be analyzed for many properties:**
- **Reachability - a marking $M_n$ is reachable from $M_0$ if there exists a firing sequence from $M_0$ to $M_n$**
  - **the set of all possible markings reachable from $M_0$ in a net $(N,M_0)$ is denoted $R(N,M_0)$ and is the set of states that the system can obtain**
- **Boundedness - a Petri Net is *k-bounded* if the number of tokens in each place does not exceed a finite number k for any marking reachable from $M_0$**
  - **by verifying that a Petri Net is *k-bounded*, it is guaranteed that any buffers of size *k* will not overflow**

Here are some of the attributes that the Petri Net can be analyzed for. All of these attributes can be examined analytically using the reachability graph and do not require simulating or "animating" the Petri Net.

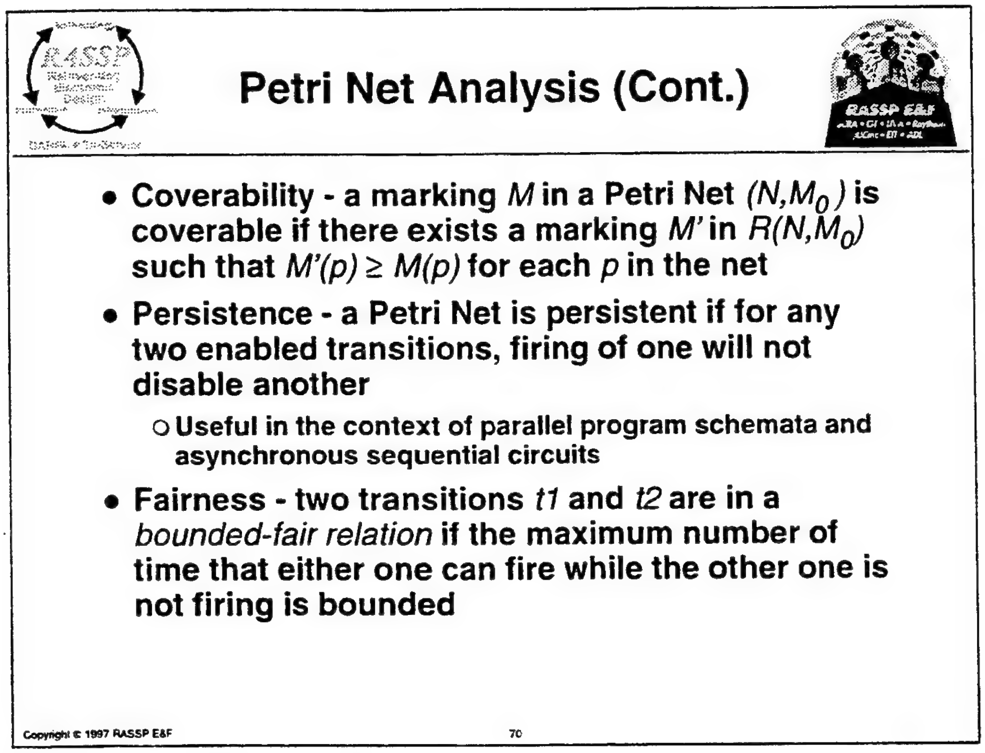
Reachability analysis can be used to see if the Petri Net can attain any "undesirable" state. Boundedness can be used to determine if the "capacity of any state (e.g. buffer size) can be overflowed.

## Petri Net Analysis (Cont.)

- **Liveness - a Petri Net *($N$,$M_0$)* is live if, no matter what marking has been reached, it is possible to fire any transition of the net through some firing sequence**
- **Liveness shows that a system has not reached a state where a portion of the system can no longer operate**
  - ○ **proving liveness is hard - so there are degrees of liveness**
- **Reversibility - a Petri Net *($N$,$M_0$)* is reversible if for each marking in *$R(N,M_0)$* it is possible to get back to *$M_0$***
- **Home state - a marking *$M'$* is a home state if it is reachable from every marking in *$R(N,M_0)$***

Liveness can again show that the Petri Net does not attain an "undesirable" state in which its not exactly deadlocked, but some transitions can no longer be fired.

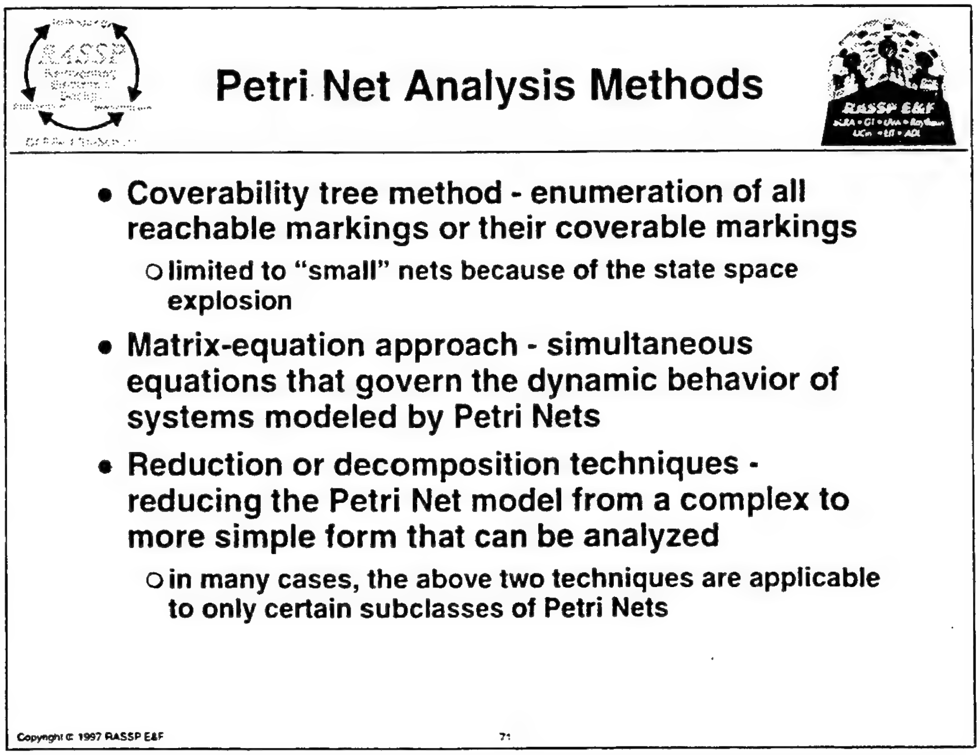
Reversibility shows that a Petri Net can regain its "home state" from any state it can attain.



## Petri Net Analysis (Cont.)

- **Coverability - a marking *M* in a Petri Net *(N,$M_0$)* is coverable if there exists a marking *M'* in *R(N,$M_0$)* such that *$M'(p) \geq M(p)$* for each *p* in the net**
- **Persistence - a Petri Net is persistent if for any two enabled transitions, firing of one will not disable another**
  - **Useful in the context of parallel program schemata and asynchronous sequential circuits**
- **Fairness - two transitions *t1* and *t2* are in a *bounded-fair relation* if the maximum number of time that either one can fire while the other one is not firing is bounded**

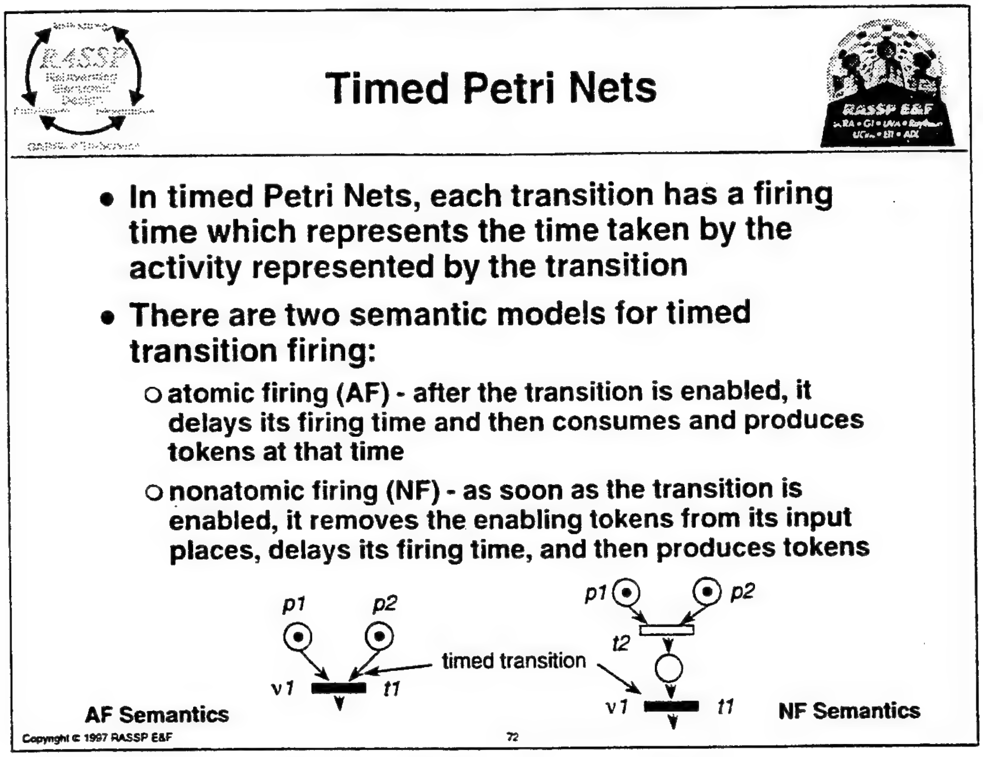
Here are more attributes that can be determined from the analysis of a Petri Net and its reachability graph.



## Petri Net Analysis Methods

- Coverability tree method - enumeration of all reachable markings or their coverable markings
  - limited to "small" nets because of the state space explosion
- Matrix-equation approach - simultaneous equations that govern the dynamic behavior of systems modeled by Petri Nets
- Reduction or decomposition techniques - reducing the Petri Net model from a complex to more simple form that can be analyzed
  - in many cases, the above two techniques are applicable to only certain subclasses of Petri Nets

Various methods for analyzing Petri Nets for the metrics discussed.



# Timed Petri Nets

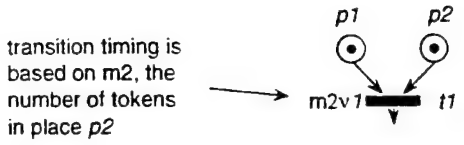
- In timed Petri Nets, each transition has a firing time which represents the time taken by the activity represented by the transition
- There are two semantic models for timed transition firing:
  - ○ atomic firing (AF) - after the transition is enabled, it delays its firing time and then consumes and produces tokens at that time
  - ○ nonatomic firing (NF) - as soon as the transition is enabled, it removes the enabling tokens from its input places, delays its firing time, and then produces tokens

Timed Petri Nets are the more useful form for performance analysis. Both NF and AF semantics can be employed although AF is more general in that NF can be described in AF.

A potential problem with AF is that in conflicting transitions, an enabled transition may be disabled during its delay time by the firing of another transition.

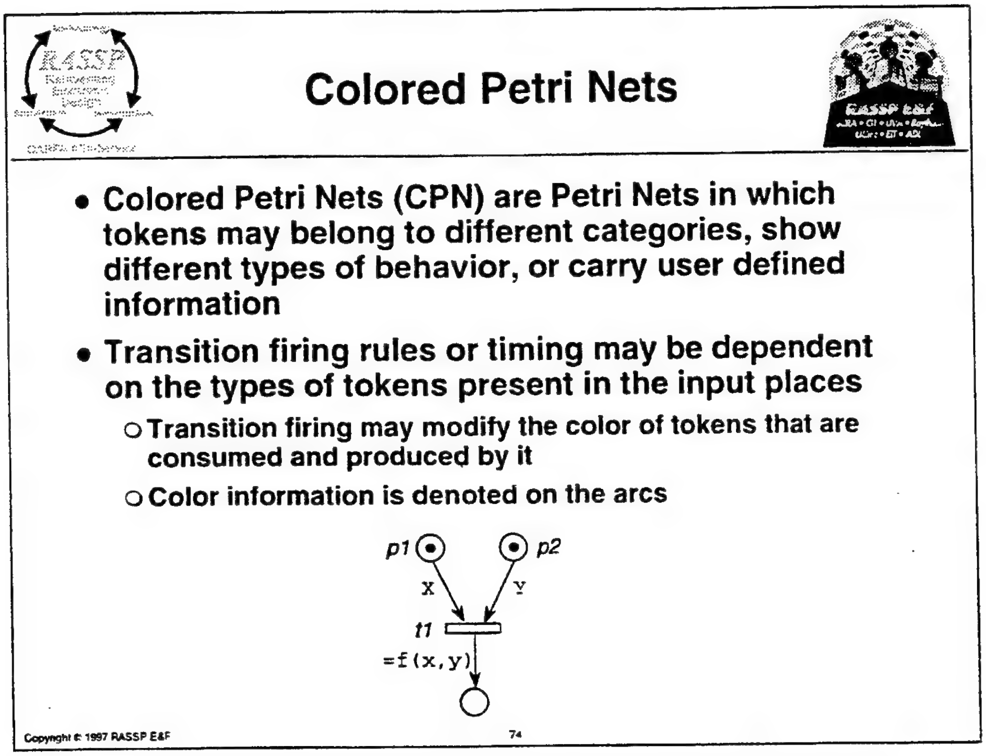
# Petri Net Timing Functions

- Transition timing functions can depend on the number of tokens in a specific place in the Petri Net


transition timing is based on m2, the number of tokens in place *p2*

- Transition timing functions can be deterministic or stochastic
- Transition timing functions can be continuous time or discrete time

Timing functions for transitions can be a function of the number of tokens in a place. Also, timing functions can be deterministic of stochastic. General Stochastic Petri Nets can be analyzed as Markov Models (as will be shown).



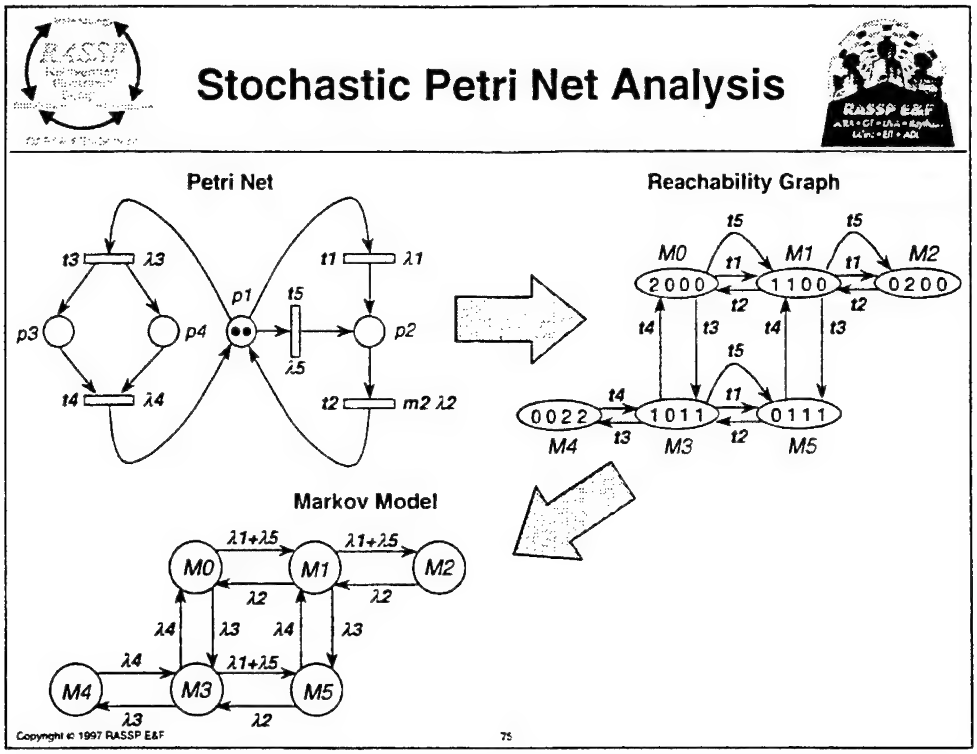
## Colored Petri Nets

- Colored Petri Nets (CPN) are Petri Nets in which tokens may belong to different categories, show different types of behavior, or carry user defined information
- Transition firing rules or timing may be dependent on the types of tokens present in the input places
  - Transition firing may modify the color of tokens that are consumed and produced by it
  - Color information is denoted on the arcs

74

Colored Petri Nets (CPN) include the notion of values (or classes) on the tokens. Note that CPNs are what is used as the mathematical foundation for UVa's ADEPT tool.

In this example, the color of the token produced by the firing of transition t1 is a function [f(x,y)] of the color of the tokens in the p1 and p2 places.



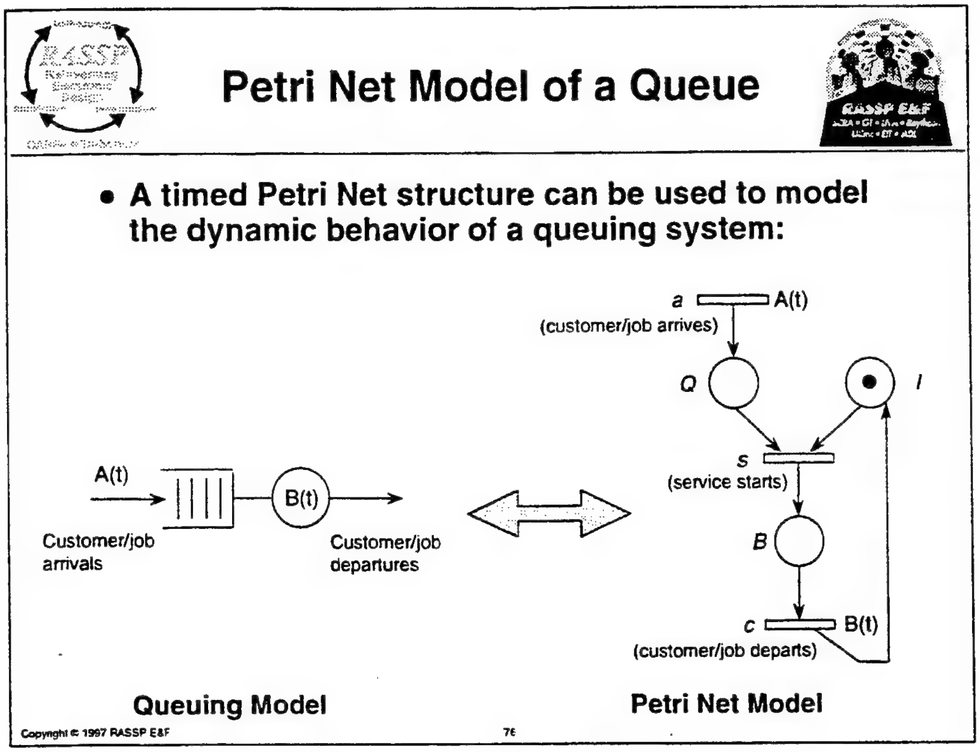
## Stochastic Petri Net Analysis

**Petri Net**

**Reachability Graph**

**Markov Model**

As shown here, a Stochastic Petri Net can be translated into a Markov Model via its reachability graph.



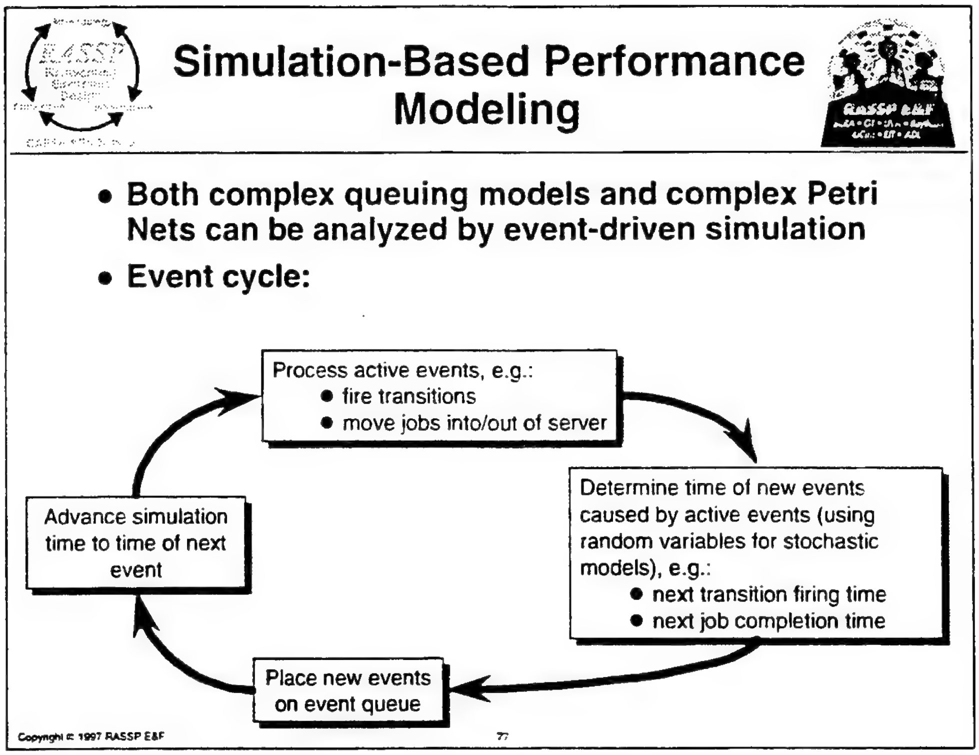
# Petri Net Model of a Queue

- **A timed Petri Net structure can be used to model the dynamic behavior of a queuing system:**

**Queuing Model**

**Petri Net Model**

Here is an example of how a queuing model can be modeled using Petri Nets - a further demonstration of their modeling power.



## Simulation-Based Performance Modeling

- **Both complex queuing models and complex Petri Nets can be analyzed by event-driven simulation**
- **Event cycle:**

Process active events, e.g.:
- fire transitions
- move jobs into/out of server

Determine time of new events caused by active events (using random variables for stochastic models), e.g.:
- next transition firing time
- next job completion time

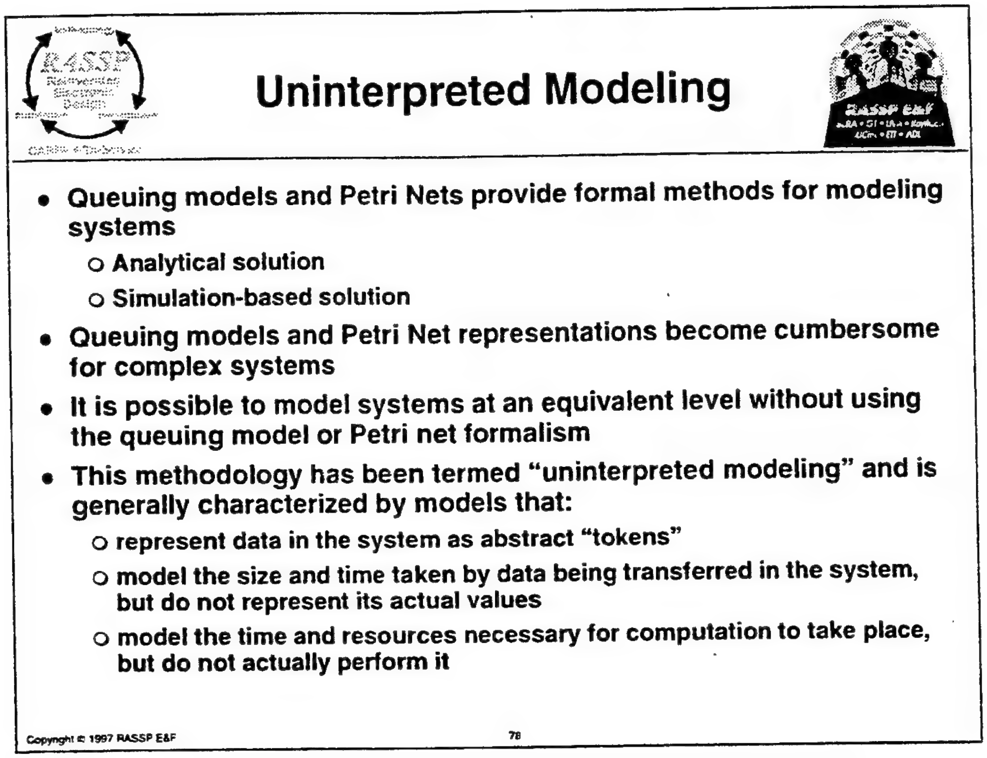
Place new events on event queue

Advance simulation time to time of next event

As mentioned before, complex queuing models and Petri Nets, although they may not be solvable via analytical techniques, can be solved by simulation. There are many commercial tools available that do this.

This is an illustration of the basic event driven simulation cycle. You simply process all events scheduled for a given time, and determine what new events are generated for what future times. These events are added to the "event queue" and time is advanced to the earliest future time in the event queue. All events at that time are then processed and the cycle begins again.

Alternatively to event-driven simulation, the simulation cycle can be done on a discrete time interval (e.g. 1 ns) and simulation time advances at regular intervals. All signals can be updated to new values (which may be the same as old ones) at each time interval. This eases the management of simulation time and the event queue.

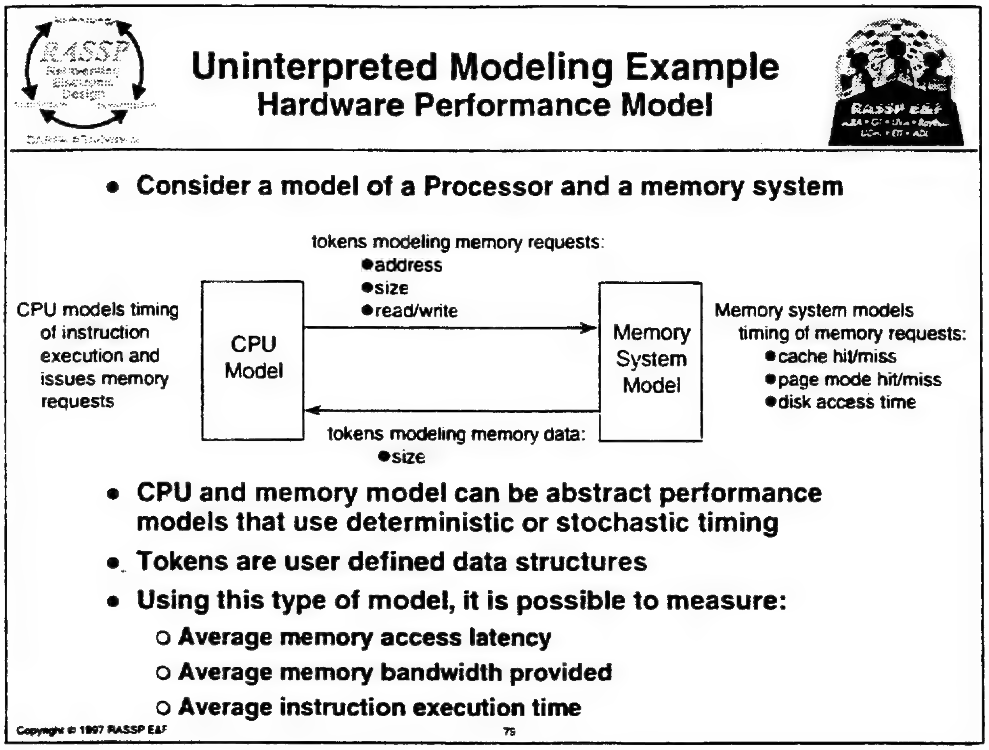


## Uninterpreted Modeling

- Queuing models and Petri Nets provide formal methods for modeling systems
  - Analytical solution
  - Simulation-based solution
- Queuing models and Petri Net representations become cumbersome for complex systems
- It is possible to model systems at an equivalent level without using the queuing model or Petri net formalism
- This methodology has been termed "uninterpreted modeling" and is generally characterized by models that:
  - represent data in the system as abstract "tokens"
  - model the size and time taken by data being transferred in the system, but do not represent its actual values
  - model the time and resources necessary for computation to take place, but do not actually perform it

It is possible to model systems at a high level without using either the queuing model or Petri Net formalism. This is a separate issue from the analytical vs. simulation-based solution issue, although models that do not have the queuing model or Petri Net formalism obviously have to use simulation-based solutions.

In general "uninterpreted modeling" the system is modeled at such a level as the data in the system that is moved from component to component is modeled, but its values and transformations performed on it are not. Timing is modeled, but usually at a high level. Recall that the taxonomy of performance models showed this level of abstraction. In general, all of the modeling environments discussed from her on out will be general "uninterpreted modeling" environments although some of them may include elements of queuing models (SES Workbench) and Petri Nets (ADEPT)



## Uninterpreted Modeling Example
### Hardware Performance Model

- **Consider a model of a Processor and a memory system**

CPU models timing of instruction execution and issues memory requests

CPU Model

tokens modeling memory requests:
- address
- size
- read/write

Memory System Model

Memory system models timing of memory requests:
- cache hit/miss
- page mode hit/miss
- disk access time

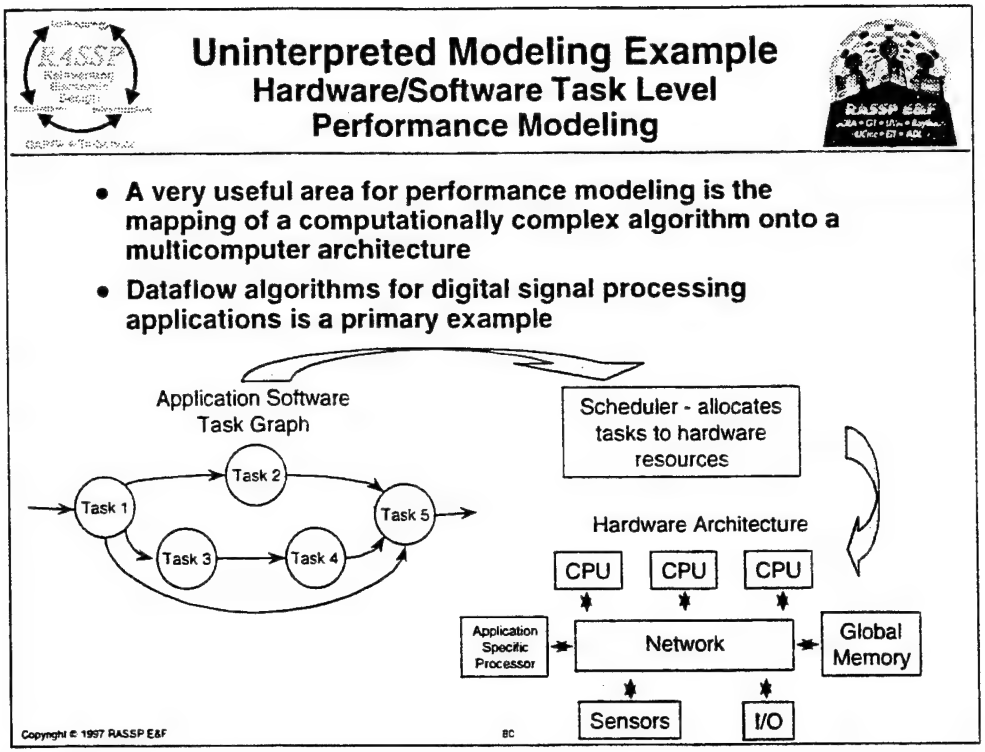
tokens modeling memory data:
- size

- **CPU and memory model can be abstract performance models that use deterministic or stochastic timing**
- **Tokens are user defined data structures**
- **Using this type of model, it is possible to measure:**
  - **Average memory access latency**
  - **Average memory bandwidth provided**
  - **Average instruction execution time**

Copyright © 1997 RASSP E&F

79

Here is an example of an uninterpreted model of a CPU and memory system. This is an example that will be utilized in the section on VHDL performance modeling examples. Notice that the tokens in the model actually model the passing of data between the CPU and the memory and are fairly abstract in nature, as are the CPU and memory component models.

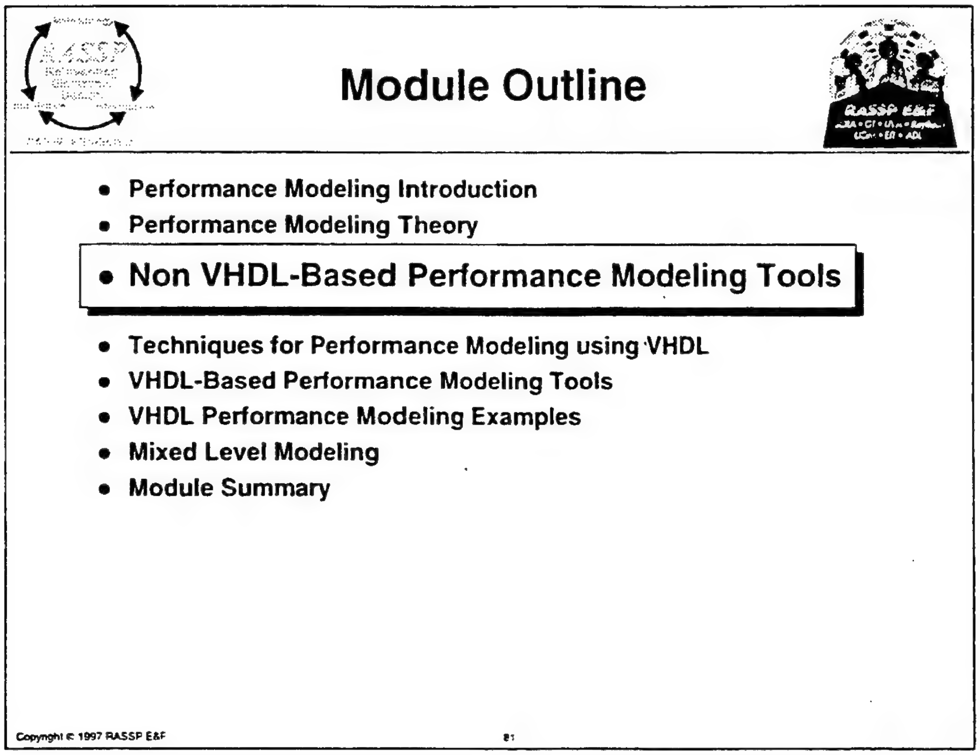


## Uninterpreted Modeling Example
### Hardware/Software Task Level Performance Modeling

- A very useful area for performance modeling is the mapping of a computationally complex algorithm onto a multicomputer architecture
- Dataflow algorithms for digital signal processing applications is a primary example

Application Software Task Graph
Task 1
Task 2
Task 3
Task 4
Task 5
Scheduler - allocates tasks to hardware resources
Hardware Architecture
CPU
CPU
CPU
Application Specific Processor
Network
Global Memory
Sensors
I/O

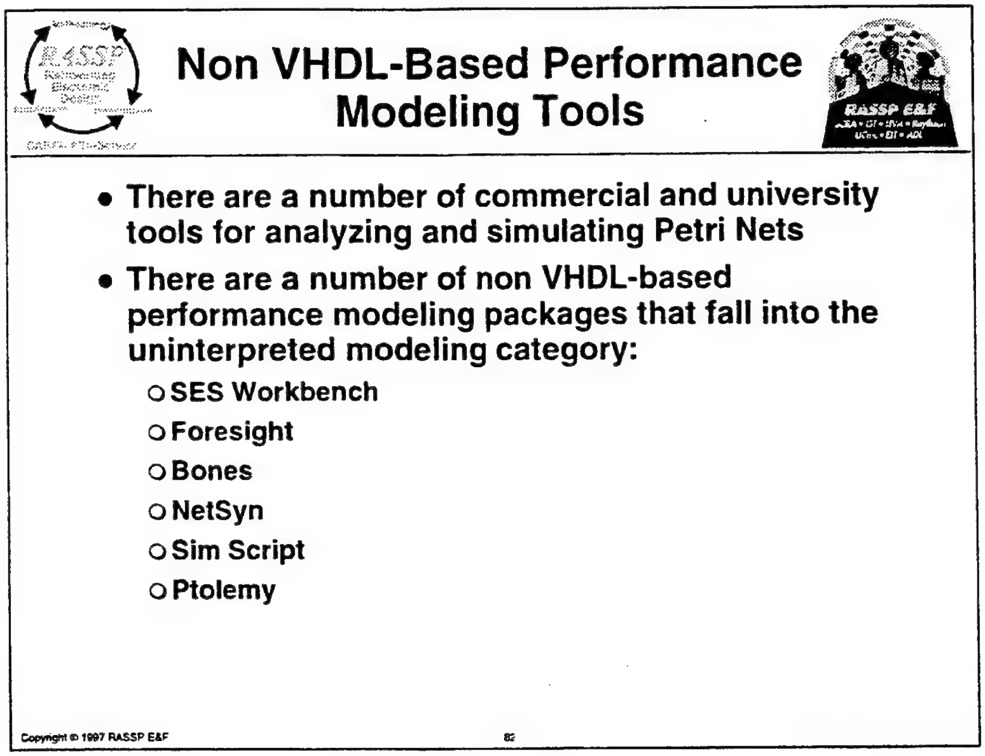
This is another type of uninterpreted model that will also be used in the example section, a hardware/software task level model. Here the software is a set of tasks, often modeled as a dataflow graph, that communicates with a "scheduler" to obtain hardware resources (processors, memories, switches) on which to execute. Usually, the software tasks provide information on how much hardware resources they require (data size, number of floating point instructions, etc.) and the hardware model actually delays the required simulated time.



# Module Outline

- Performance Modeling Introduction
- Performance Modeling Theory
- **Non VHDL-Based Performance Modeling Tools**
- Techniques for Performance Modeling using VHDL
- VHDL-Based Performance Modeling Tools
- VHDL Performance Modeling Examples
- Mixed Level Modeling
- Module Summary

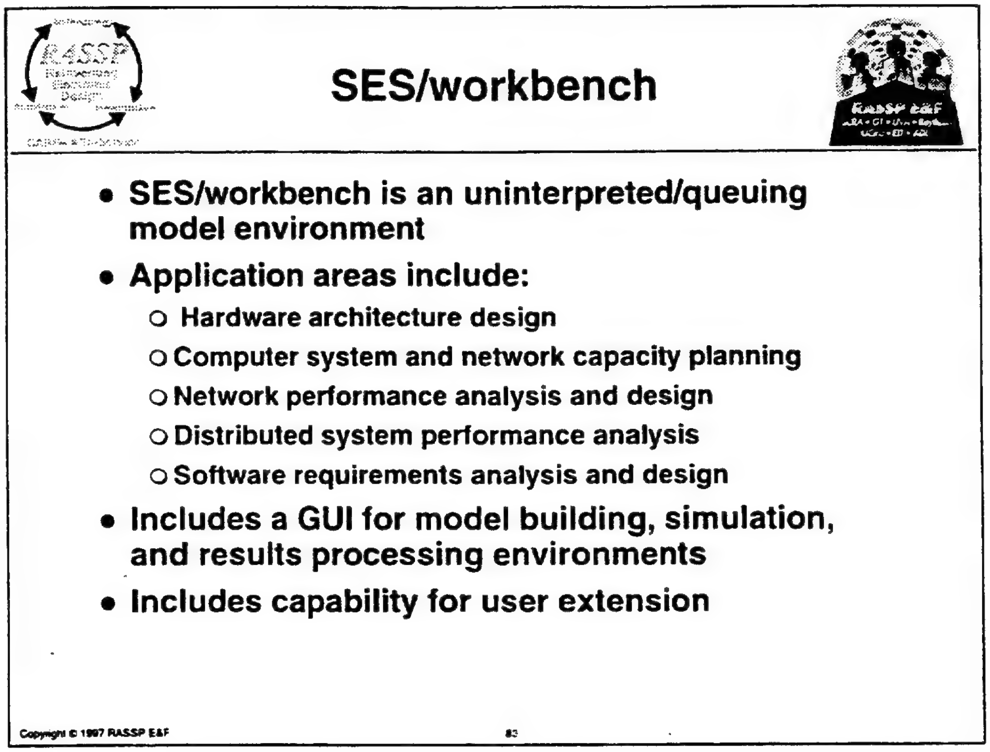
Module Outline



## Non VHDL-Based Performance Modeling Tools

- There are a number of commercial and university tools for analyzing and simulating Petri Nets
- There are a number of non VHDL-based performance modeling packages that fall into the uninterpreted modeling category:
    - ○ SES Workbench
    - ○ Foresight
    - ○ Bones
    - ○ NetSyn
    - ○ Sim Script
    - ○ Ptolemy

There are a number of commercial and educational packages available for Petri Net analysis and general "uninterpreted" performance modeling. Most of these are implemented in C or C++ and as such, are a bit divorced from the electronic system design process. However, because of their number and popularity, some discussion of them is warranted here.



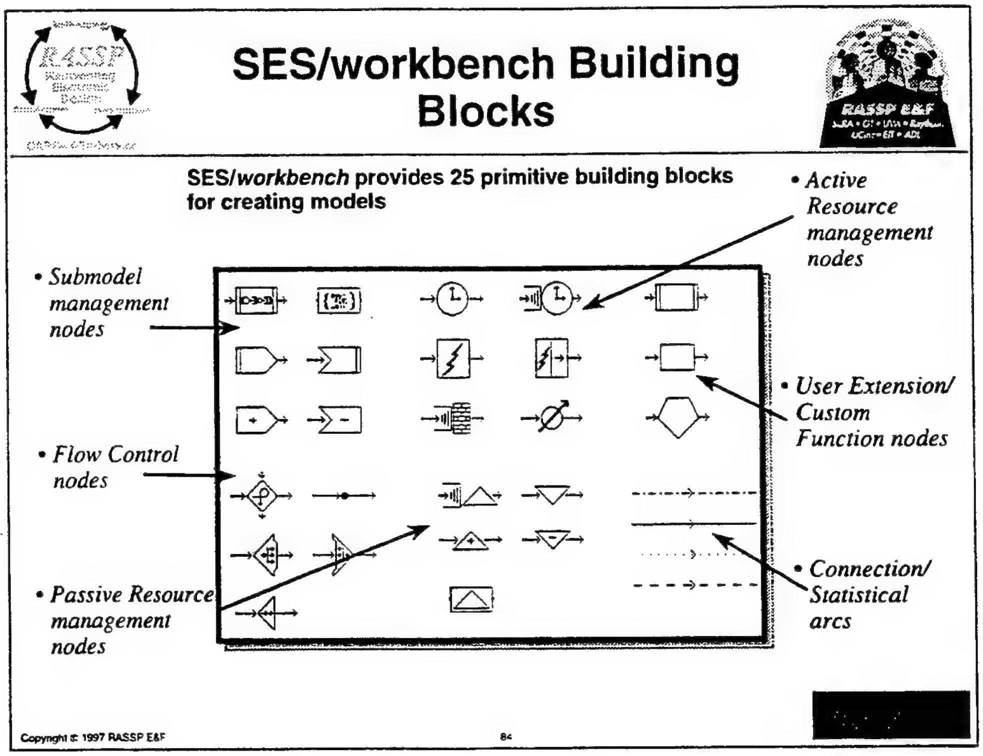
## SES/workbench

- SES/workbench is an uninterpreted/queuing model environment
- Application areas include:
  - ○ Hardware architecture design
  - ○ Computer system and network capacity planning
  - ○ Network performance analysis and design
  - ○ Distributed system performance analysis
  - ○ Software requirements analysis and design
- Includes a GUI for model building, simulation, and results processing environments
- Includes capability for user extension

As an example of the types of tools in the general uninterpreted performance modeling category that are available, SES workbench will be presented in some detail. SES does have some basis in queuing network modeling, but performance models that do not include queues can be built with it, so it falls into the more general category.

This presentation was taken from the Scientific and Engineering Software, Inc. web page: http://www.ses.com

A through reading of the material on Workbench there will suffice as background to present these slides.

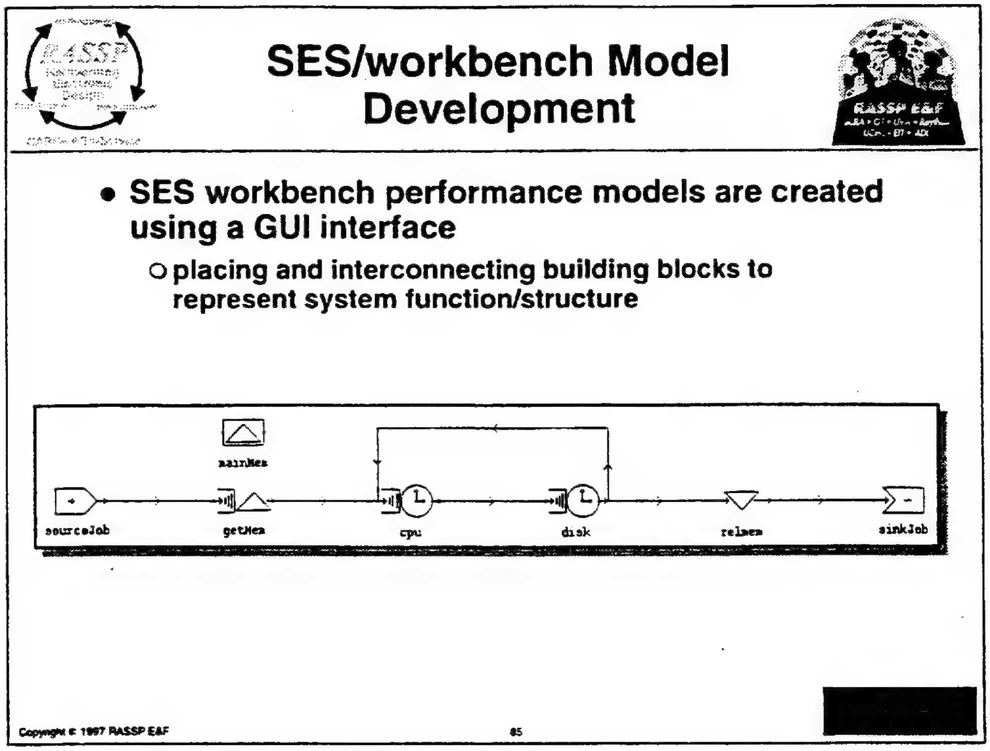


# SES/workbench Building Blocks

**SES/*workbench* provides 25 primitive building blocks for creating models**

- *Submodel management nodes*
- *Flow Control nodes*
- *Passive Resource management nodes*
- *Active Resource management nodes*
- *User Extension/ Custom Function nodes*
- *Connection/ Statistical arcs*

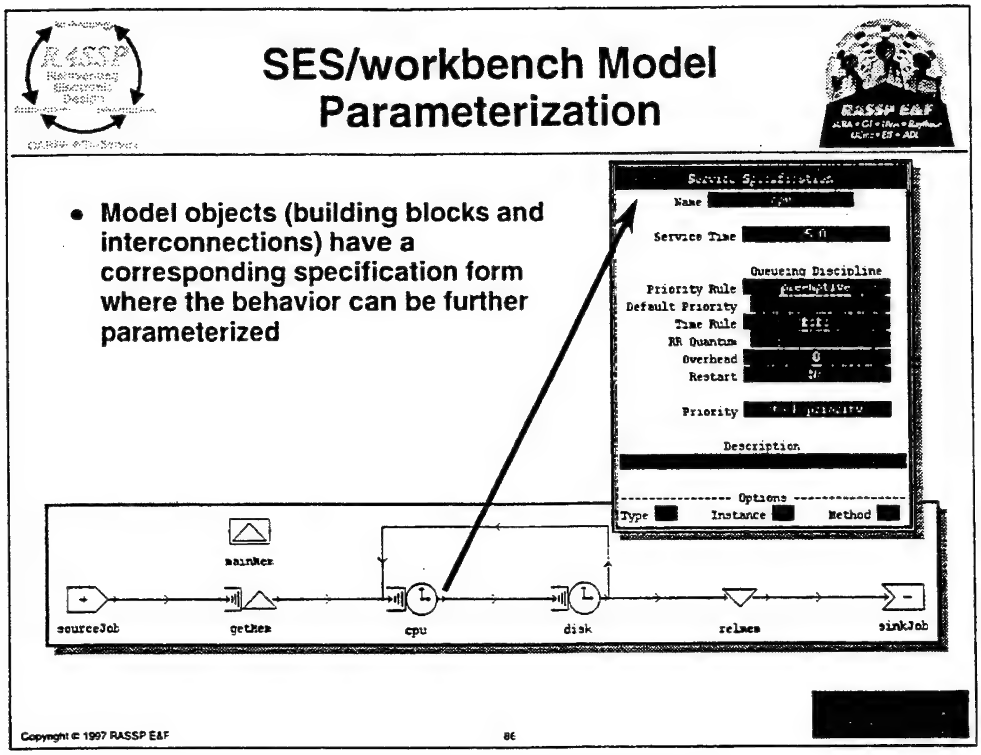
See http://www.ses.com


mainMem
sourceJob
getMem
cpu
disk
relmem
sinkJob

# SES/workbench Model Development

- SES workbench performance models are created using a GUI interface
  - ○ placing and interconnecting building blocks to represent system function/structure

See http://www.ses.com



# SES/workbench Model Parameterization

- Model objects (building blocks and interconnections) have a corresponding specification form where the behavior can be further parameterized

Service Specification

Name [illegible]

Service Time 5.0

Queueing Discipline

Priority Rule [illegible]

Default Priority

Time Rule [illegible]

RR Quantum

Overhead 0

Restart [illegible]

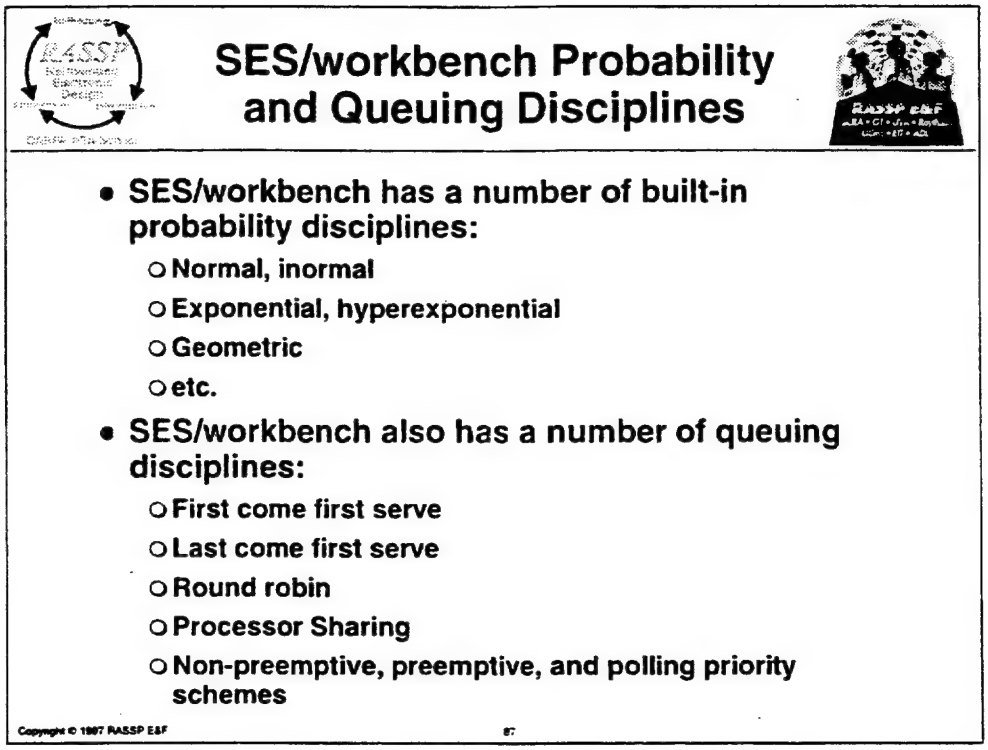
Priority [illegible]

Description

Options

Type ■ Instance ■ Method ■

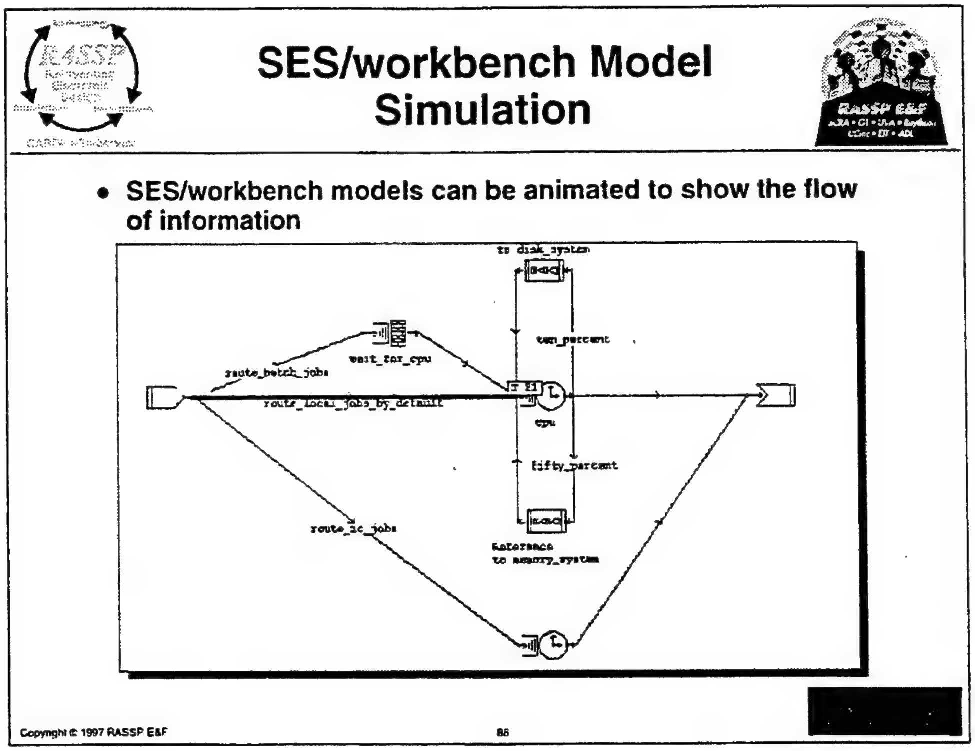
See http://www.ses.com



## SES/workbench Probability and Queuing Disciplines

- SES/workbench has a number of built-in probability disciplines:
  - ○ Normal, inormal
  - ○ Exponential, hyperexponential
  - ○ Geometric
  - ○ etc.
- SES/workbench also has a number of queuing disciplines:
  - ○ First come first serve
  - ○ Last come first serve
  - ○ Round robin
  - ○ Processor Sharing
  - ○ Non-preemptive, preemptive, and polling priority schemes

See http://www.ses.com



## SES/workbench Model Simulation

- SES/workbench models can be animated to show the flow of information

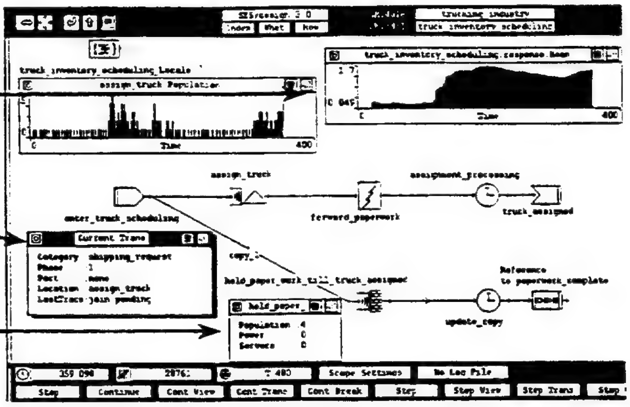
[redacted]

See http://www.ses.com

# SES/workbench Model Simulation (Cont.)

- **SES/workbench includes the capability of viewing the model statistics as the model executes**

- *Gather statistics on workload, environment and application performance*
  - *Inspect the current work in your system*
  - *Analyze the application load on the execution environment*



Category shipping_request
Phase 1
Port none
Location assign_truck
LastTrace join pending

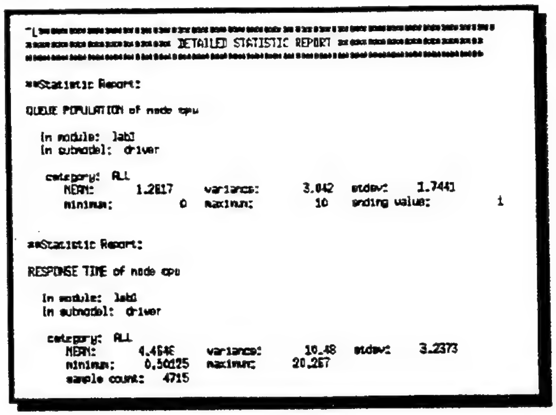
Population 4
Power 0
Servers 0

[redacted]

See http://www.ses.com

# SES/workbench Model Simulation (Cont.)

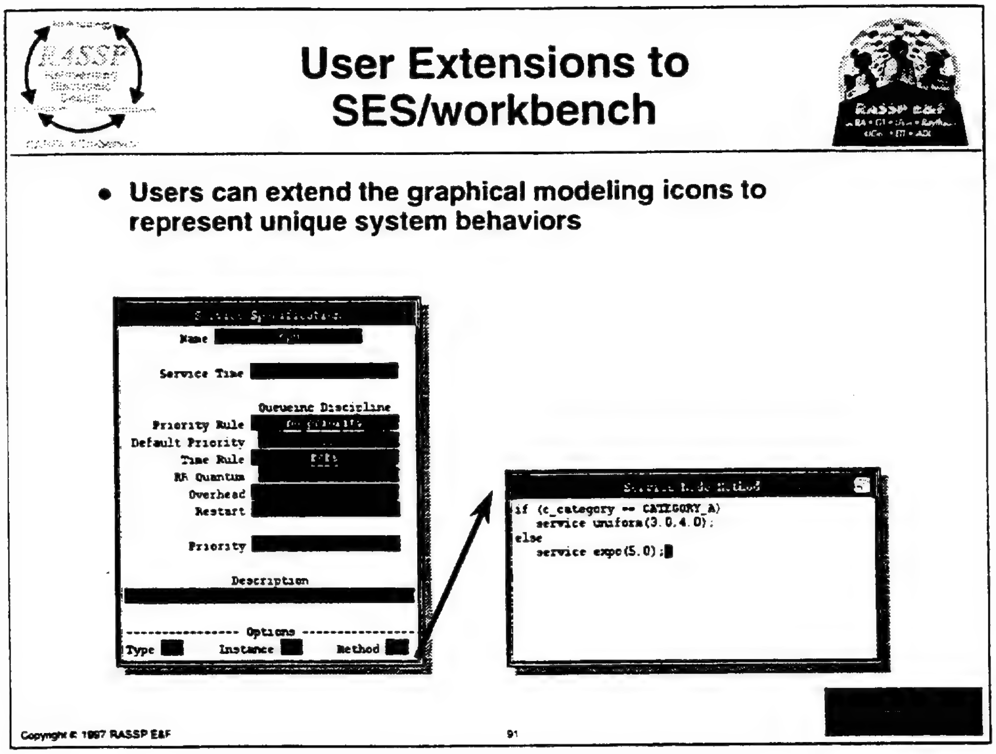
- SES/workbench provides model statistics on system performance that permit verification, debugging, and optimization of system designs
- Statistics may be built-in or user-defined



```
DETAILED STATISTIC REPORT

**Statistic Report:

QUEUE POPULATION of node cpu

  In module:  lab1
  In submodel:  driver

   category:  ALL
     MEAN:       1.2817     variance:      3.042   stdev:    1.7441
     minimum:         0     maximum:         10   ending value:          1

**Statistic Report:

RESPONSE TIME of node cpu

  In module:  lab1
  In submodel:  driver

   category:  ALL
     MEAN:       4.4646     variance:      10.48   stdev:    3.2373
     minimum:   0.50125     maximum:     20.267
     sample count:   4715
```

See http://www.ses.com

23



# User Extensions to SES/workbench

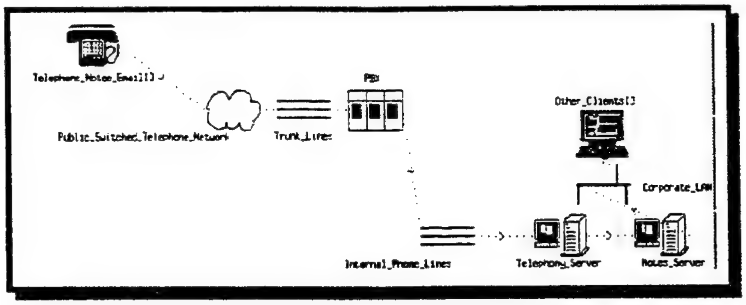
- **Users can extend the graphical modeling icons to represent unique system behaviors**

```
if (c_category == CATEGORY_A)
   service uniform(3.0,4.0);
else
   service expo(5.0);
```

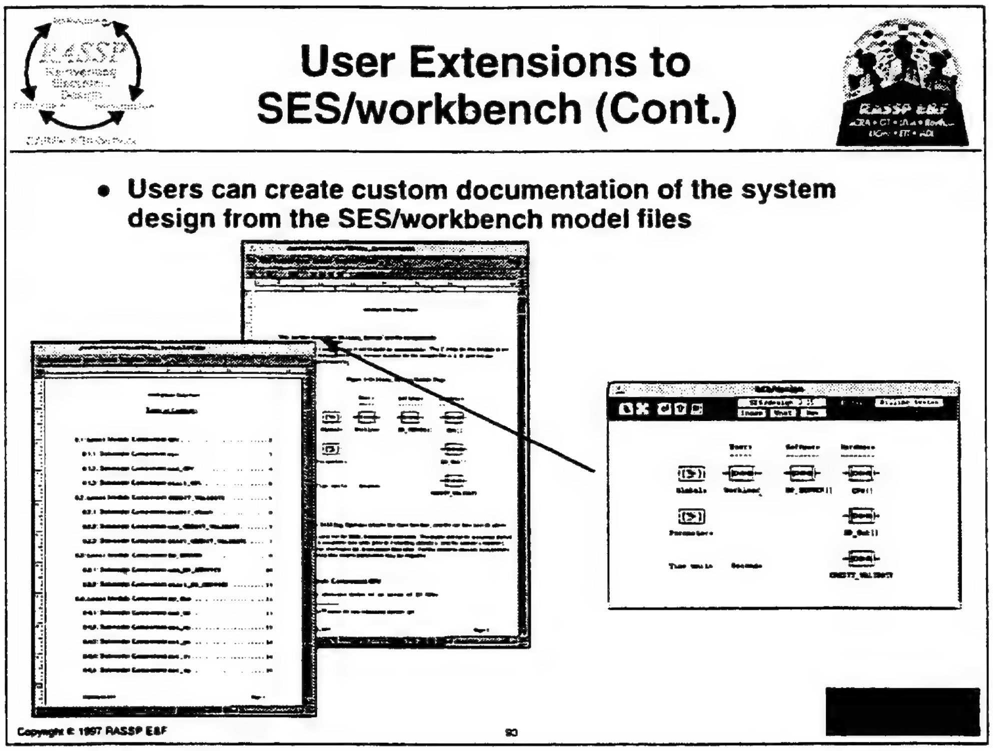
See http://www.ses.com

# User Extensions to SES/workbench (Cont.)

- **Users can add custom icons to the SES/workbench to represent portions of the modeled system in a more self-explanatory manner**



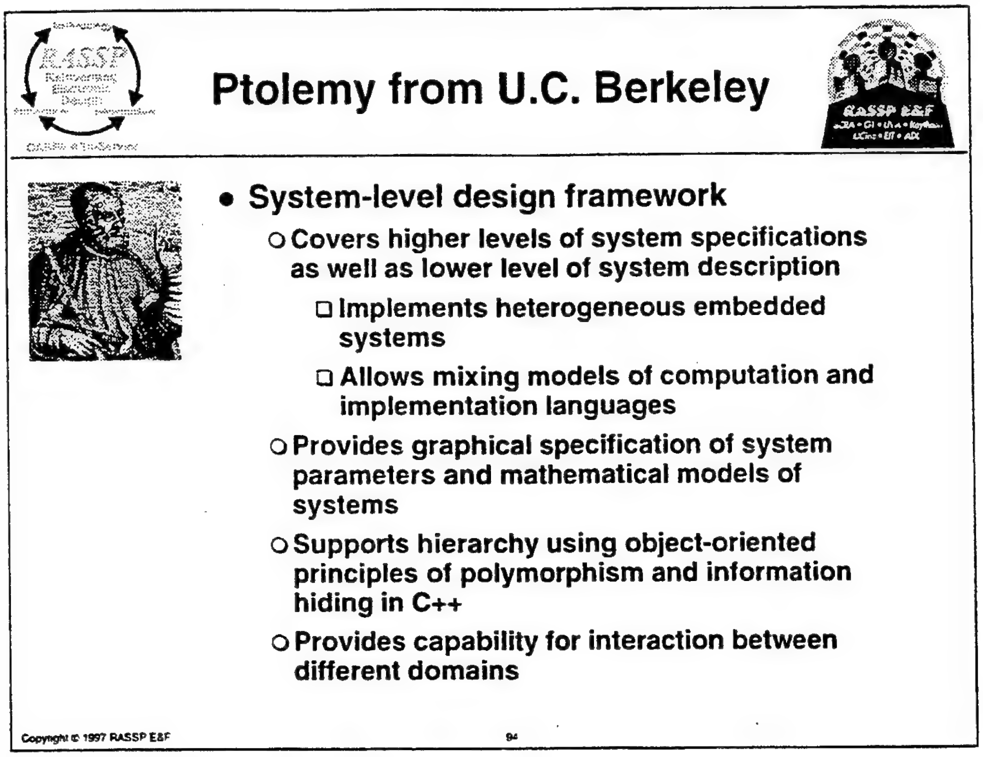
See http://www.ses.com



## User Extensions to SES/workbench (Cont.)

- Users can create custom documentation of the system design from the SES/workbench model files

See http://www.ses.com



## Ptolemy from U.C. Berkeley

- **System-level design framework**
  - ○ **Covers higher levels of system specifications as well as lower level of system description**
    - □ **Implements heterogeneous embedded systems**
    - □ **Allows mixing models of computation and implementation languages**
  - ○ **Provides graphical specification of system parameters and mathematical models of systems**
  - ○ **Supports hierarchy using object-oriented principles of polymorphism and information hiding in C++**
  - ○ **Provides capability for interaction between different domains**

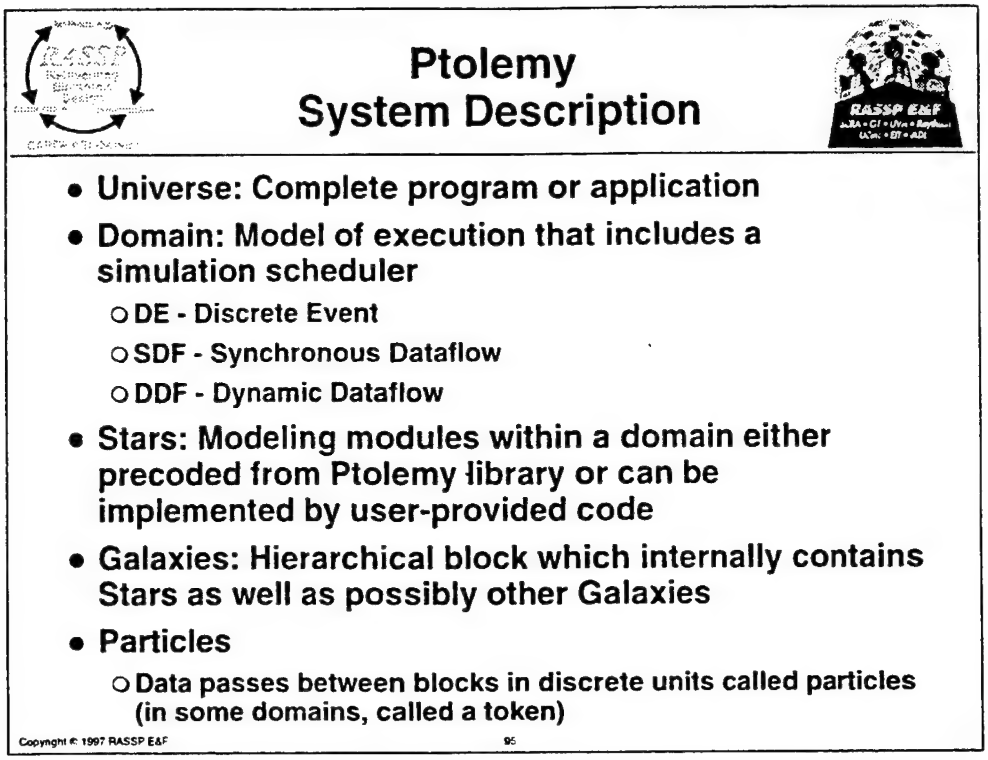
[Ptolemy96].

This section describes UC Berkeley's Ptolemy functional modeling tool. Ptolemy is targeted as a tool to model and simulate the function of a DSP system, but, as is described in this section, it has been used to perform uninterpreted performance modeling.

Biographical Names

Ptol-e-my \'ta^:l-e-me^-\

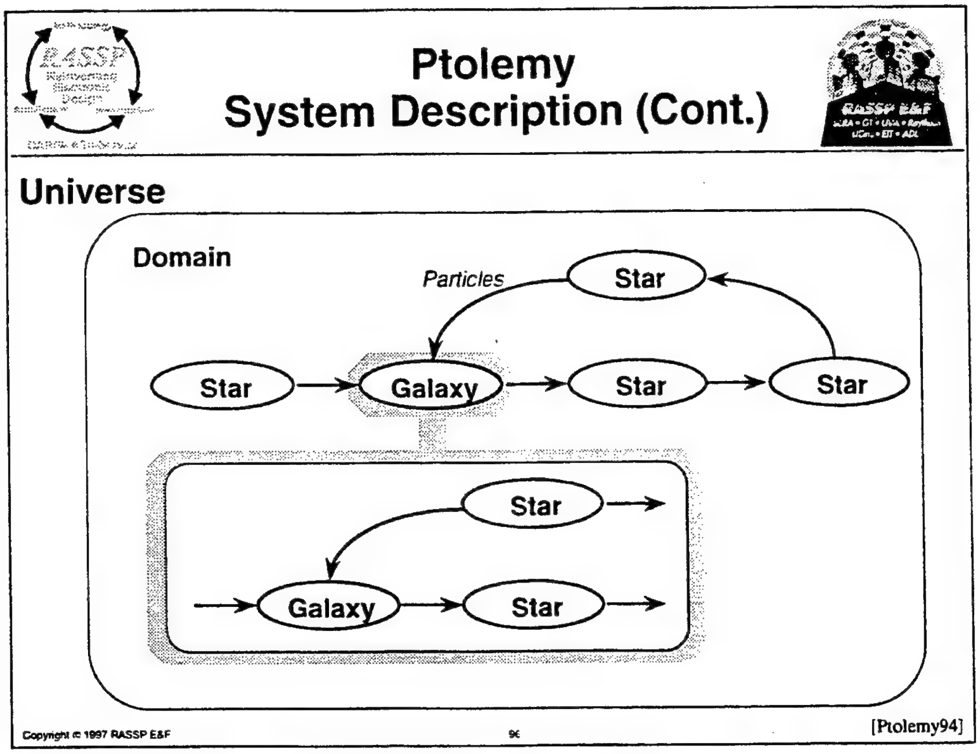
2d cent. A.D. Claudius Ptolemaeus - Alexandrian astronomer



## Ptolemy System Description

- Universe: Complete program or application
- Domain: Model of execution that includes a simulation scheduler
  - DE - Discrete Event
  - SDF - Synchronous Dataflow
  - DDF - Dynamic Dataflow
- Stars: Modeling modules within a domain either precoded from Ptolemy library or can be implemented by user-provided code
- Galaxies: Hierarchical block which internally contains Stars as well as possibly other Galaxies
- Particles
  - Data passes between blocks in discrete units called particles (in some domains, called a token)

This slide outlines the parts of a Polemy simulation.



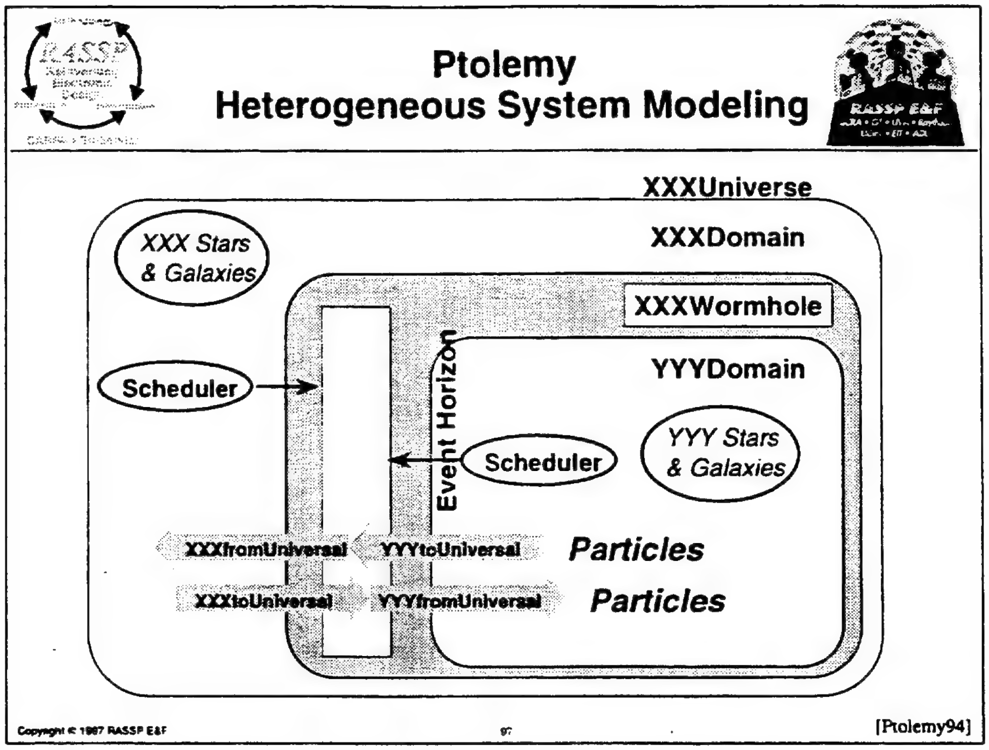
## Ptolemy
## System Description (Cont.)

**Universe**

96

[Ptolemy94]

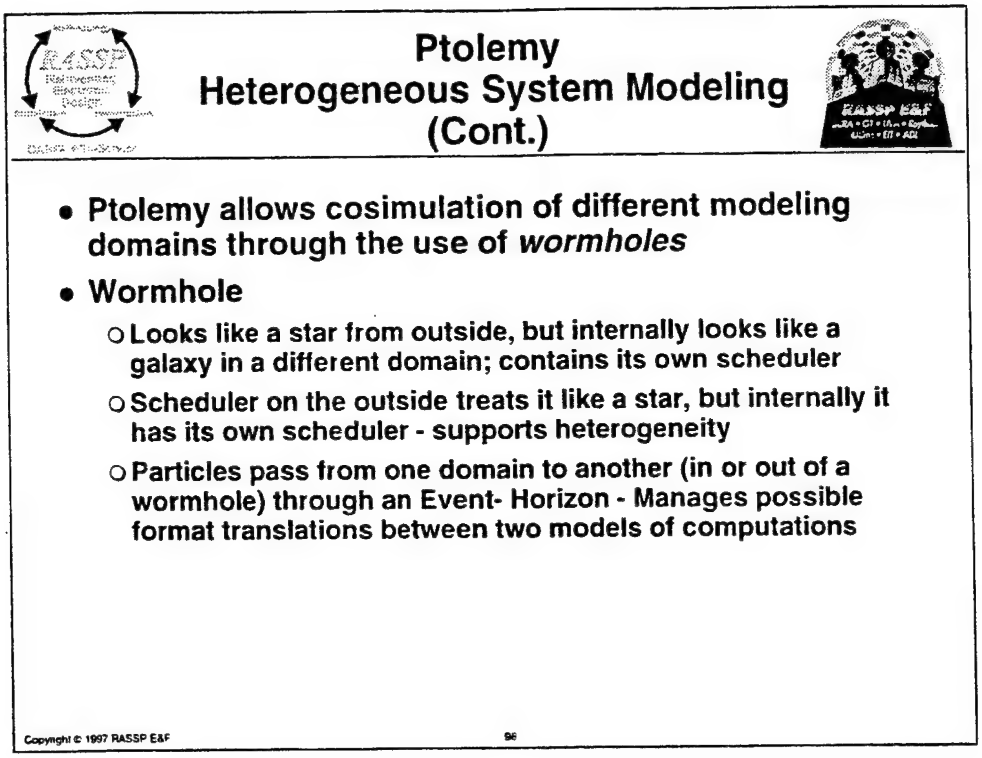
This figure shows the general outline of a system model in Ptolemy. General modeling blocks in Ptolemy are called "stars." A hierarchical collection of stars used to model a large piece of functionality is called a Galaxy. Stars communicate with each other by passing particles (similar to tokens). A specific modeling paradigm in Ptolemy is called a domain. An entire model is Ptolemy is called a Universe.



## Ptolemy
## Heterogeneous System Modeling

[Ptolemy94]

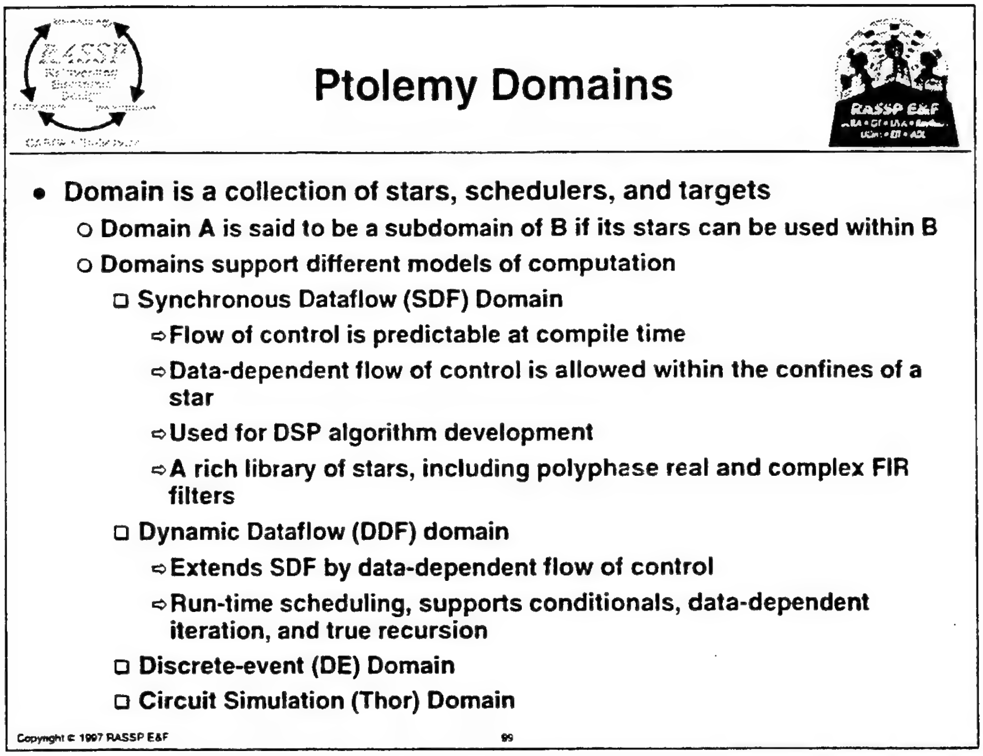
A model of computation (such as discrete event, synchronous dataflow, dynamic dataflow, etc.) is called a Domain in Ptolemy. Each domain includes building blocks, or stars (which the user can add to by writing their own), a scheduler that executes the portion of a model that resides in its domain, and wormholes that interface data and events between domains.



## Ptolemy Heterogeneous System Modeling (Cont.)

- **Ptolemy allows cosimulation of different modeling domains through the use of *wormholes***
- **Wormhole**
  - **Looks like a star from outside, but internally looks like a galaxy in a different domain; contains its own scheduler**
  - **Scheduler on the outside treats it like a star, but internally it has its own scheduler - supports heterogeneity**
  - **Particles pass from one domain to another (in or out of a wormhole) through an Event- Horizon - Manages possible format translations between two models of computations**

Stars communicate across different domains using wormholes. Wormholes allow heterogeneous models with stars from different domains to be constructed.



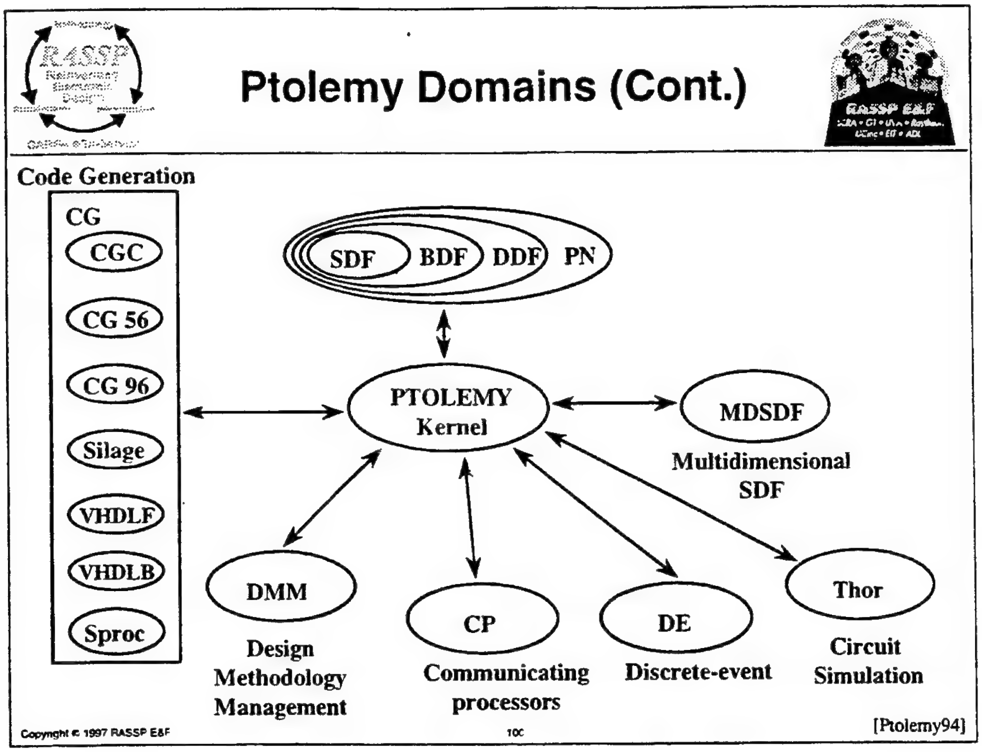
## Ptolemy Domains

- Domain is a collection of stars, schedulers, and targets
  - ○ Domain A is said to be a subdomain of B if its stars can be used within B
  - ○ Domains support different models of computation
    - ☐ Synchronous Dataflow (SDF) Domain
      - ⇨ Flow of control is predictable at compile time
      - ⇨ Data-dependent flow of control is allowed within the confines of a star
      - ⇨ Used for DSP algorithm development
      - ⇨ A rich library of stars, including polyphase real and complex FIR filters
    - ☐ Dynamic Dataflow (DDF) domain
      - ⇨ Extends SDF by data-dependent flow of control
      - ⇨ Run-time scheduling, supports conditionals, data-dependent iteration, and true recursion
    - ☐ Discrete-event (DE) Domain
    - ☐ Circuit Simulation (Thor) Domain

More discussion of domains.

Example:

> A high-level dataflow model of a signal processing system can be connected to a hardware simulator that in turn may be connected to a discrete-event model of a communication network
>
> BDF domain implements a compile-time scheduler for DDF graphs that supports run-time flow of control; similar to SDF. Attempts to construct a compile-time scheduler - like DDF
>
> - achieves the efficiency of SDF with the generality of DDF.
>
> HOF domain: takes a function as an argument and/or returns a function. It implements a star called Map, that can apply any other star (or galaxy) to the sequence(s) at its inputs thereby “mapping” itself to the other star or galaxy.

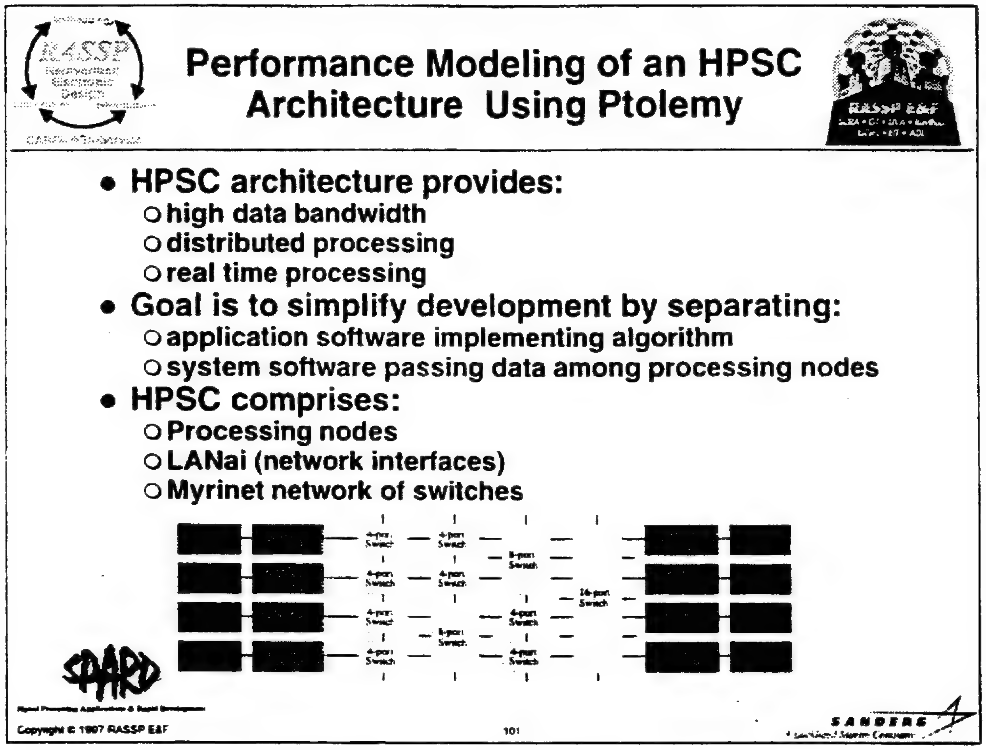


## Ptolemy Domains (Cont.)

10C

[Ptolemy94]

This is a graphical representation of the domains available within Ptolemy and how they interact with the Ptolemy kernel.



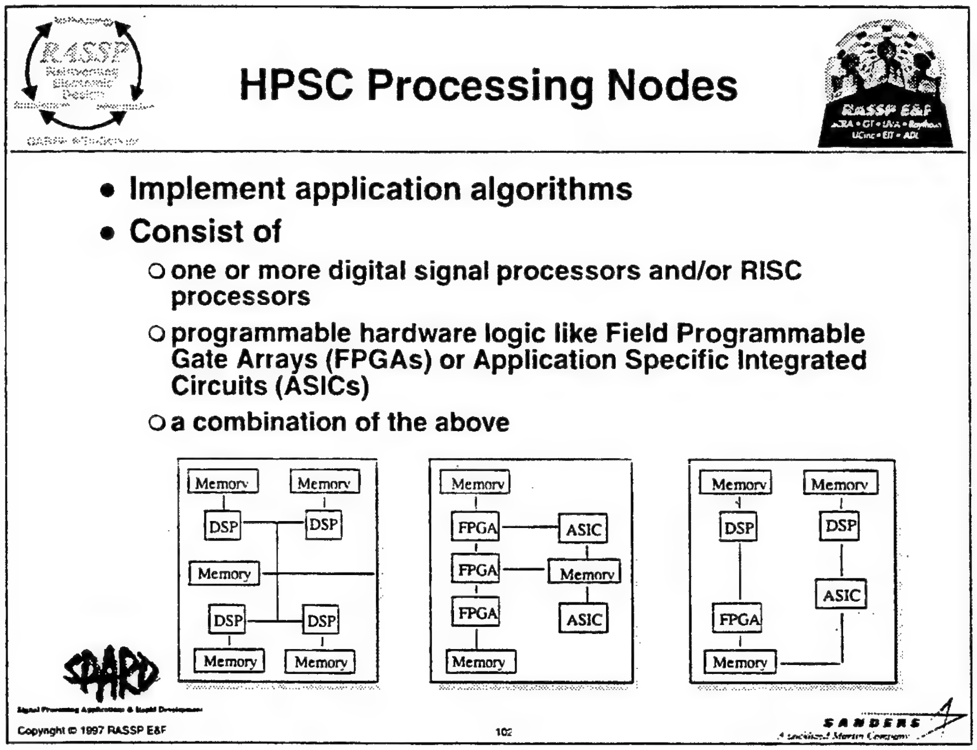
4-port Switch
8-port Switch
16-port Switch

## Performance Modeling of an HPSC Architecture Using Ptolemy

- **HPSC architecture provides:**
  - high data bandwidth
  - distributed processing
  - real time processing
- **Goal is to simplify development by separating:**
  - application software implementing algorithm
  - system software passing data among processing nodes
- **HPSC comprises:**
  - Processing nodes
  - LANai (network interfaces)
  - Myrinet network of switches

SANDERS
A Lockheed Martin Company

This is a presentation of how High Performance Scalable Computing systems can be accomplished using Ptolemy. HPSC systems are those types of systems utilized in the RASSP program. This method for performance modeling is described in detail in [Pauer97], so a through reading of that paper will suffice to explain these slides.



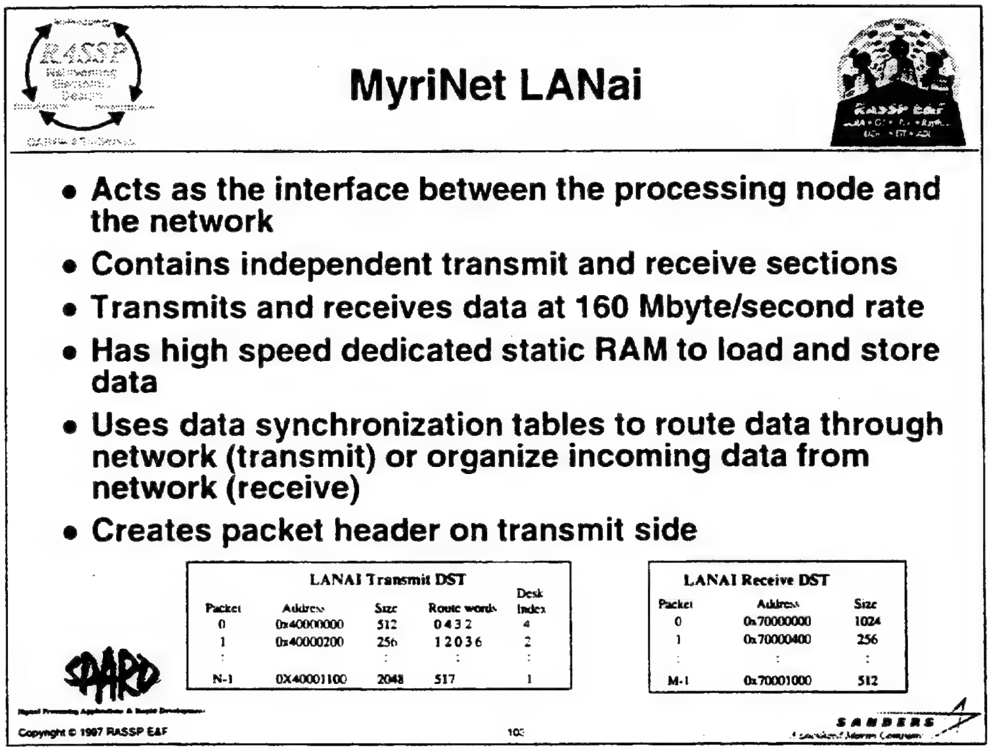
## HPSC Processing Nodes

- **Implement application algorithms**
- **Consist of**
  - ○ one or more digital signal processors and/or RISC processors
  - ○ programmable hardware logic like Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs)
  - ○ a combination of the above

Memory
Memory
DSP
DSP
Memory
DSP
DSP
Memory
Memory
Memory
FPGA
ASIC
FPGA
Memory
FPGA
ASIC
Memory
Memory
Memory
DSP
DSP
ASIC
FPGA
Memory

SANDERS
A Lockheed Martin Company

See [Pauer97].



# MyriNet LANai

- Acts as the interface between the processing node and the network
- Contains independent transmit and receive sections
- Transmits and receives data at 160 Mbyte/second rate
- Has high speed dedicated static RAM to load and store data
- Uses data synchronization tables to route data through network (transmit) or organize incoming data from network (receive)
- Creates packet header on transmit side

**LANAI Transmit DST**

| Packet | Address | Size | Route words | Desk Index |
|---|---|---|---|---|
| 0 | 0x40000000 | 512 | 0 4 3 2 | 4 |
| 1 | 0x40000200 | 256 | 1 2 0 3 6 | 2 |
| : |  | : | : | : |
| N-1 | 0X40001100 | 2048 | 517 | 1 |

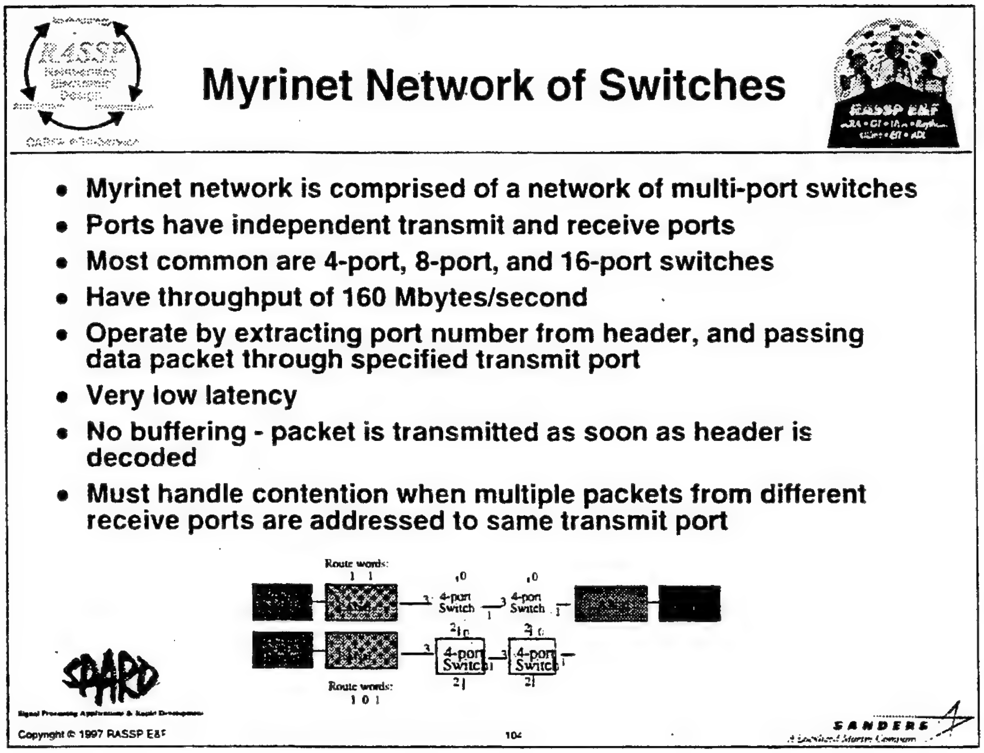
**LANAI Receive DST**

| Packet | Address | Size |
|---|---|---|
| 0 | 0x70000000 | 1024 |
| 1 | 0x70000400 | 256 |
| : | : | : |
| M-1 | 0x70001000 | 512 |

SANDERS
A Lockheed Martin Company

See [Pauer97].

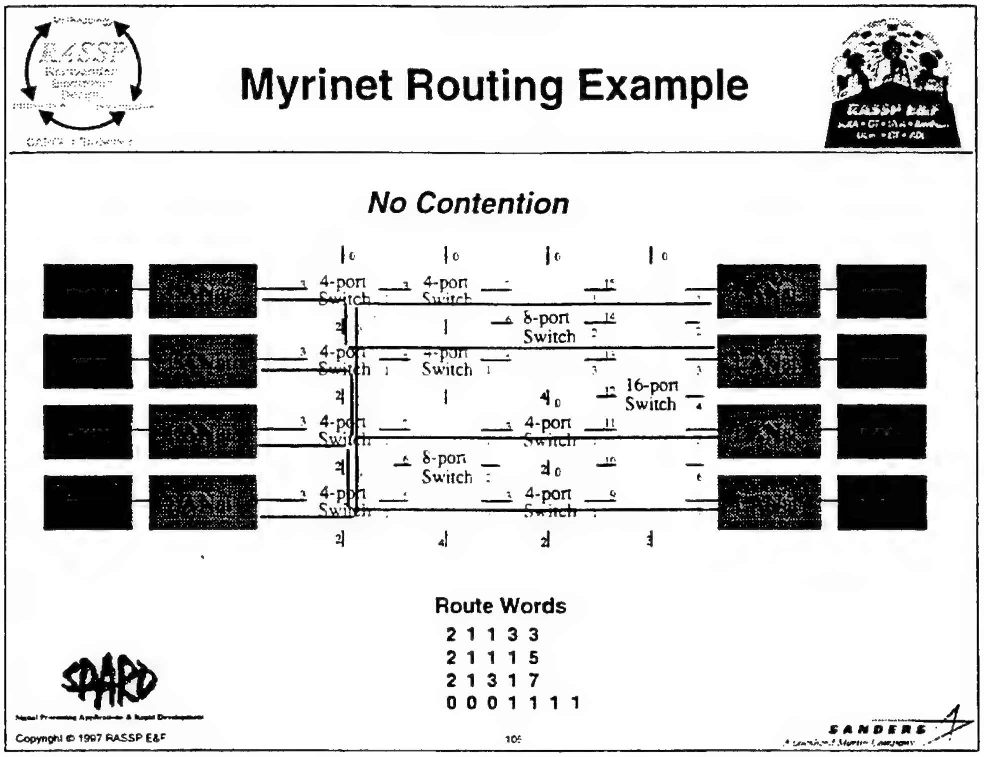
247



# Myrinet Network of Switches

- Myrinet network is comprised of a network of multi-port switches
- Ports have independent transmit and receive ports
- Most common are 4-port, 8-port, and 16-port switches
- Have throughput of 160 Mbytes/second
- Operate by extracting port number from header, and passing data packet through specified transmit port
- Very low latency
- No buffering - packet is transmitted as soon as header is decoded
- Must handle contention when multiple packets from different receive ports are addressed to same transmit port

See [Pauer97].



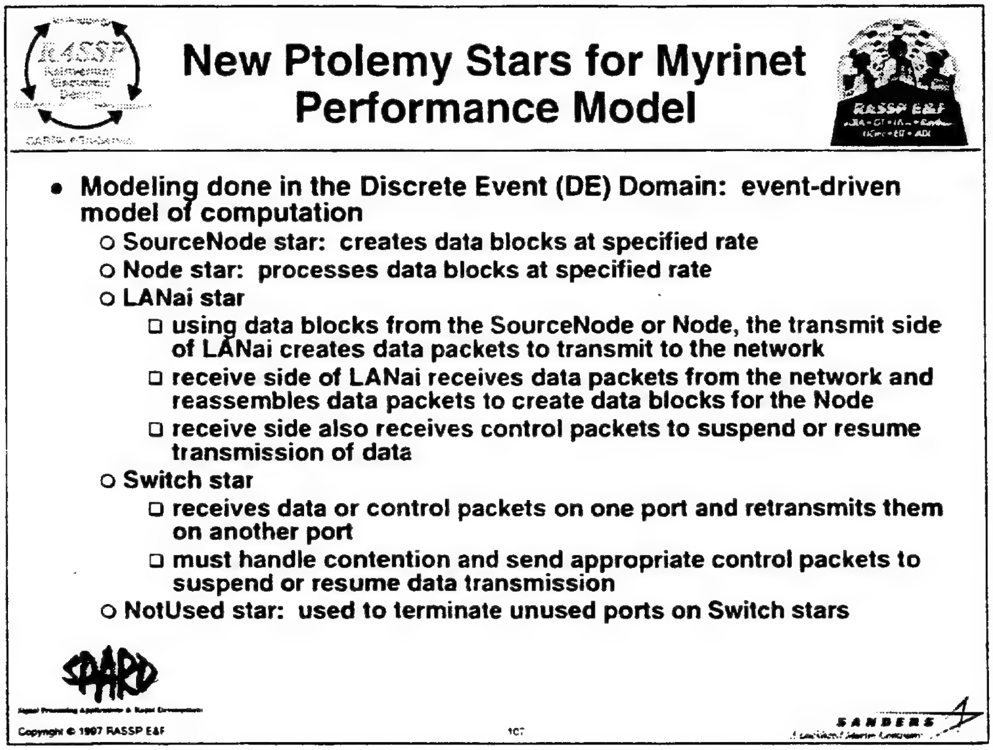
## Myrinet Routing Example

### *No Contention*

**Route Words**

2 1 1 3 3
2 1 1 1 5
2 1 3 1 7
0 0 0 1 1 1 1

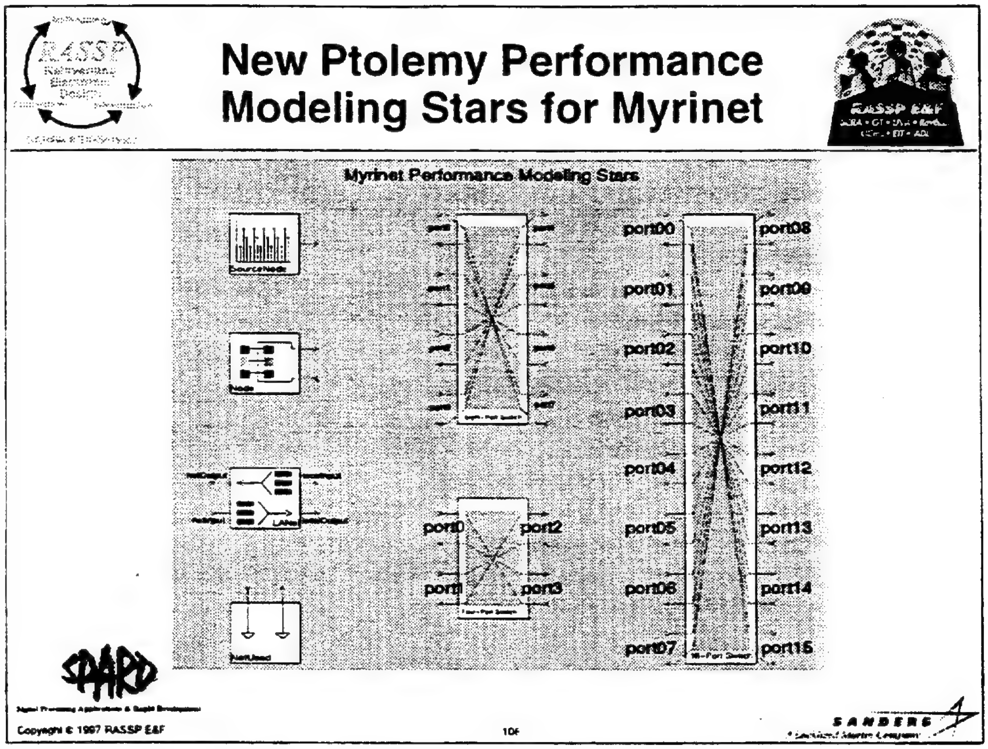
See [Pauer97].



## New Ptolemy Stars for Myrinet Performance Model

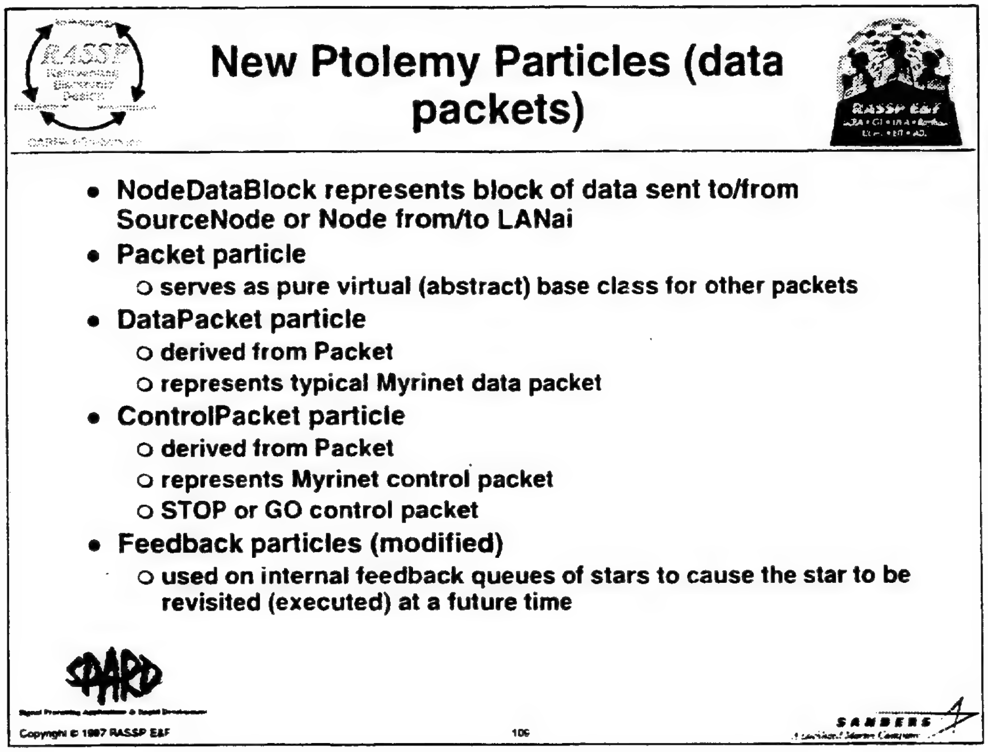
- Modeling done in the Discrete Event (DE) Domain: event-driven model of computation
  - ○ SourceNode star: creates data blocks at specified rate
  - ○ Node star: processes data blocks at specified rate
  - ○ LANai star
    - ☐ using data blocks from the SourceNode or Node, the transmit side of LANai creates data packets to transmit to the network
    - ☐ receive side of LANai receives data packets from the network and reassembles data packets to create data blocks for the Node
    - ☐ receive side also receives control packets to suspend or resume transmission of data
  - ○ Switch star
    - ☐ receives data or control packets on one port and retransmits them on another port
    - ☐ must handle contention and send appropriate control packets to suspend or resume data transmission
  - ○ NotUsed star: used to terminate unused ports on Switch stars

See [Pauer97].



## New Ptolemy Performance Modeling Stars for Myrinet

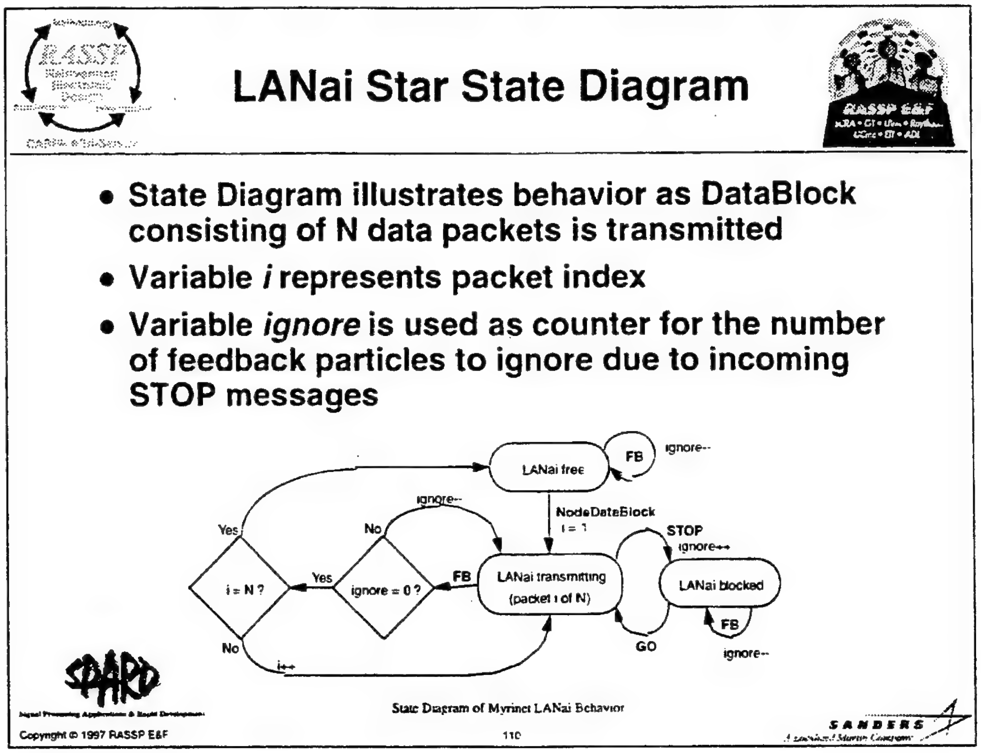
See [Pauer97].



## New Ptolemy Particles (data packets)

- NodeDataBlock represents block of data sent to/from SourceNode or Node from/to LANai
- Packet particle
  - serves as pure virtual (abstract) base class for other packets
- DataPacket particle
  - derived from Packet
  - represents typical Myrinet data packet
- ControlPacket particle
  - derived from Packet
  - represents Myrinet control packet
  - STOP or GO control packet
- Feedback particles (modified)
  - used on internal feedback queues of stars to cause the star to be revisited (executed) at a future time

See [Pauer97].



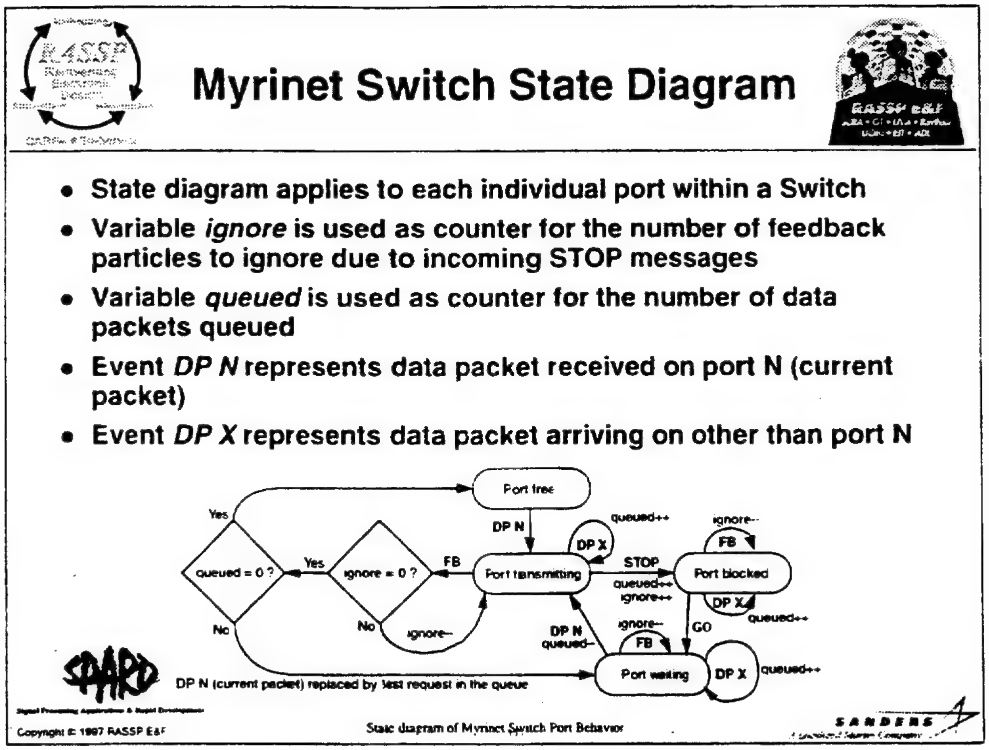
## LANai Star State Diagram

- State Diagram illustrates behavior as DataBlock consisting of N data packets is transmitted
- Variable *i* represents packet index
- Variable *ignore* is used as counter for the number of feedback particles to ignore due to incoming STOP messages

State Diagram of Myrinet LANai Behavior

SANDERS

See [Pauer97].



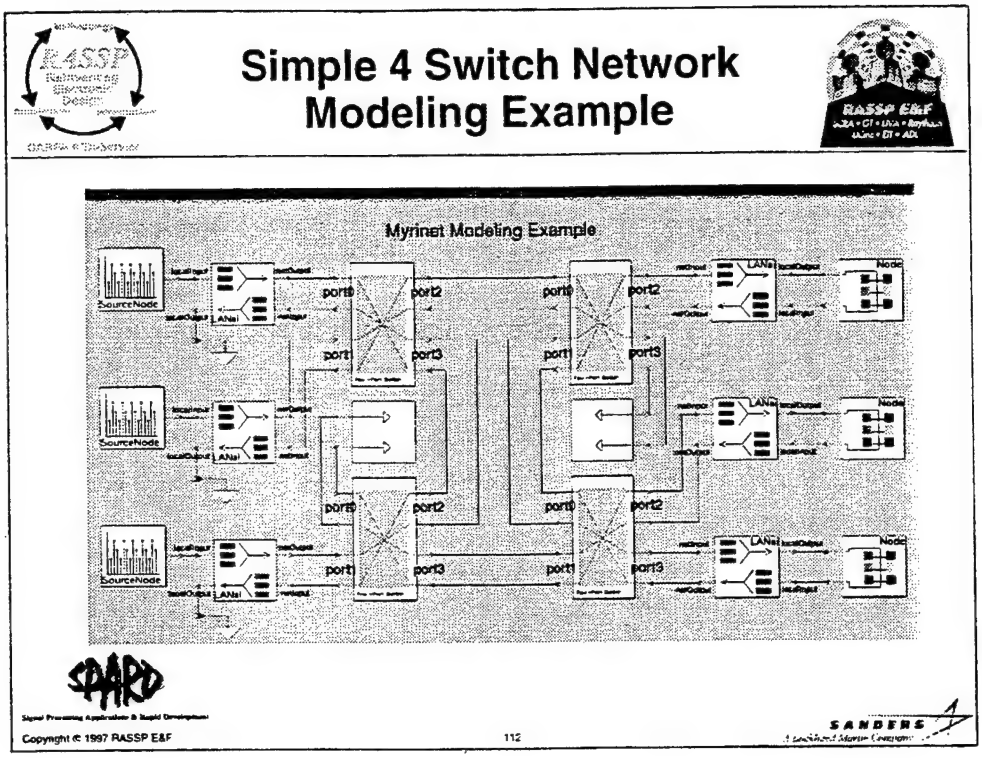
# Myrinet Switch State Diagram

- State diagram applies to each individual port within a Switch
- Variable *ignore* is used as counter for the number of feedback particles to ignore due to incoming STOP messages
- Variable *queued* is used as counter for the number of data packets queued
- Event *DP N* represents data packet received on port N (current packet)
- Event *DP X* represents data packet arriving on other than port N

State diagram of Myrinet Switch Port Behavior

SANDERS

See [Pauer97].



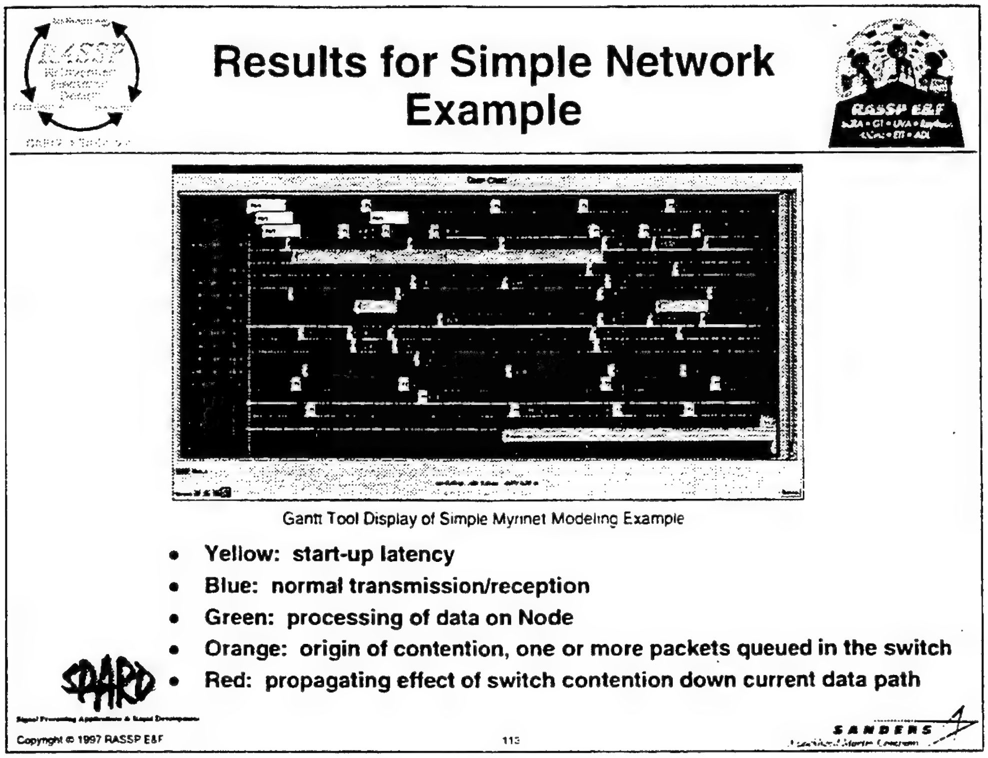
## Simple 4 Switch Network Modeling Example

Myrinet Modeling Example

SPARD
Signal Processing Applications & Rapid Development

SANDERS
A Lockheed Martin Company

See [Pauer97].



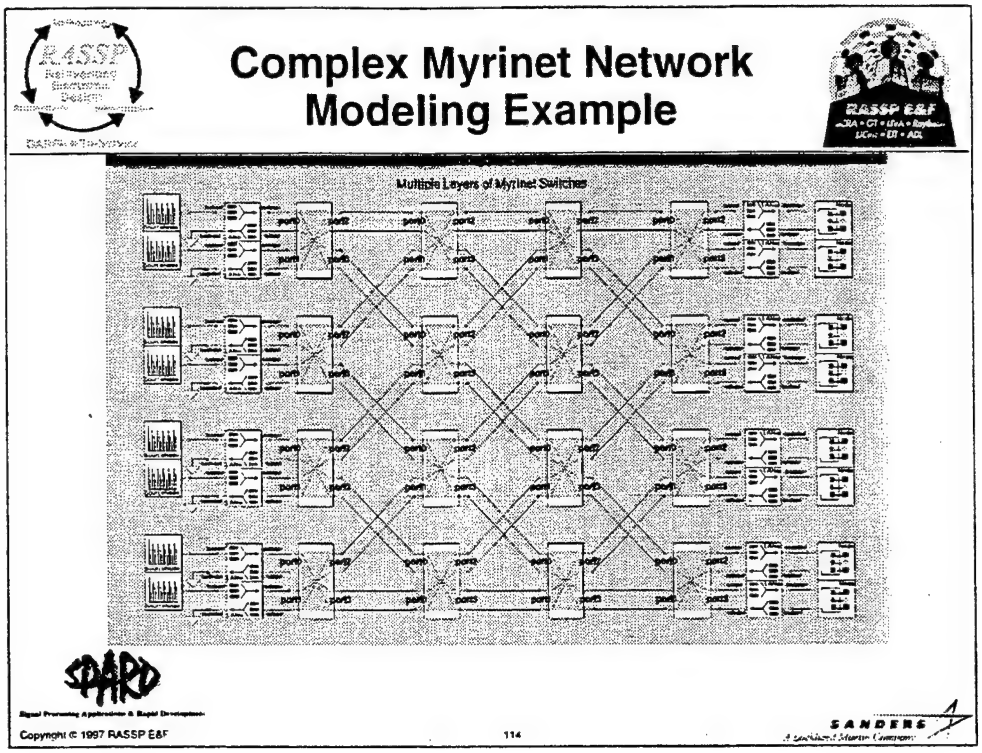
## Results for Simple Network Example

Gantt Tool Display of Simple Myrinet Modeling Example

- **Yellow: start-up latency**
- **Blue: normal transmission/reception**
- **Green: processing of data on Node**
- **Orange: origin of contention, one or more packets queued in the switch**
- **Red: propagating effect of switch contention down current data path**

SANDERS

See [Pauer97].



## Complex Myrinet Network Modeling Example

Multiple Layers of Myrinet Switches

SPARD

Signal Processing Applications & Rapid Development

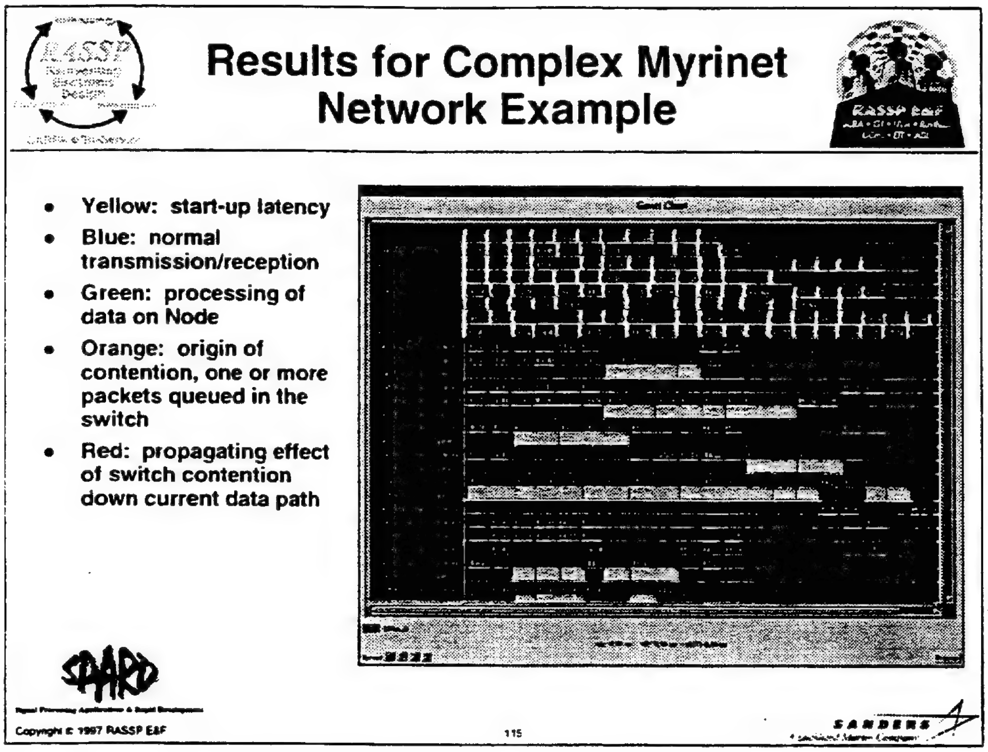
Copyright © 1997 RASSP E&F

114

SANDERS

A Lockheed Martin Company

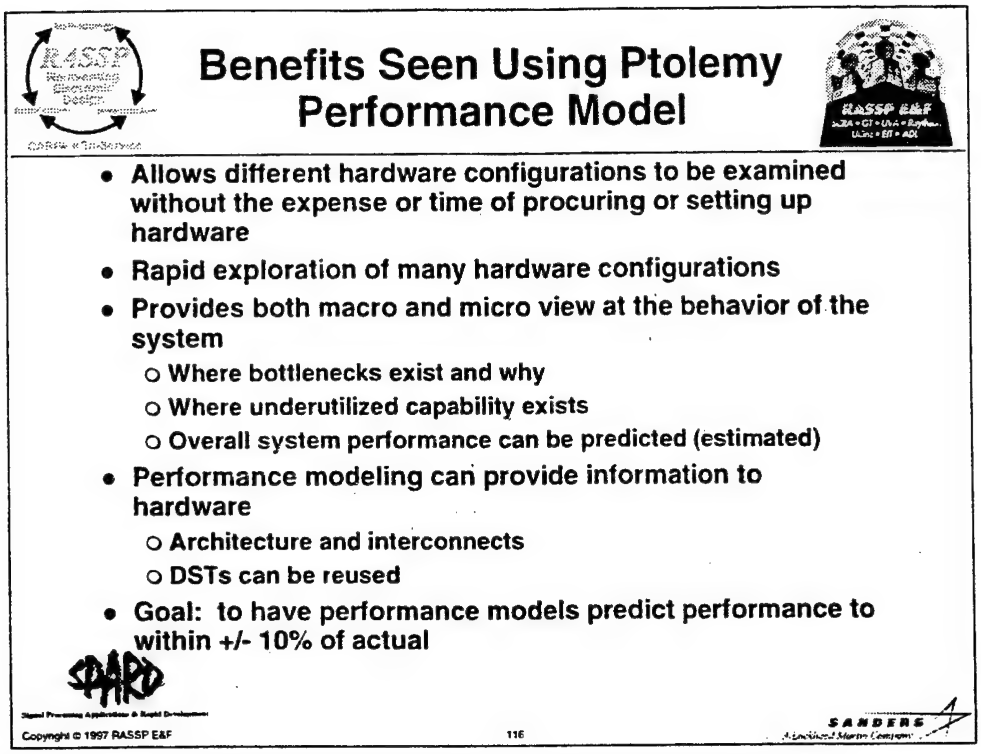
See [Pauer97].



## Results for Complex Myrinet Network Example

- Yellow: start-up latency
- Blue: normal transmission/reception
- Green: processing of data on Node
- Orange: origin of contention, one or more packets queued in the switch
- Red: propagating effect of switch contention down current data path

See [Pauer97].

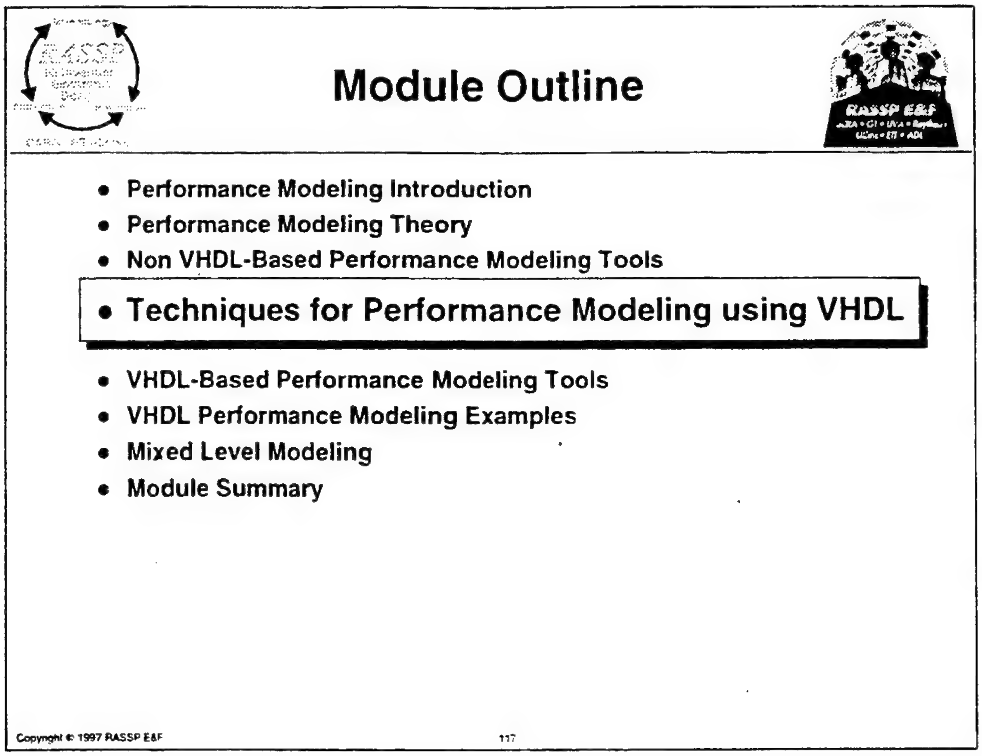


# Benefits Seen Using Ptolemy Performance Model

- Allows different hardware configurations to be examined without the expense or time of procuring or setting up hardware
- Rapid exploration of many hardware configurations
- Provides both macro and micro view at the behavior of the system
  - Where bottlenecks exist and why
  - Where underutilized capability exists
  - Overall system performance can be predicted (estimated)
- Performance modeling can provide information to hardware
  - Architecture and interconnects
  - DSTs can be reused
- Goal: to have performance models predict performance to within +/- 10% of actual

SANDERS
A Lockheed Martin Company

See [Pauer97].



## Module Outline

- Performance Modeling Introduction
- Performance Modeling Theory
- Non VHDL-Based Performance Modeling Tools
- **Techniques for Performance Modeling using VHDL**
- VHDL-Based Performance Modeling Tools
- VHDL Performance Modeling Examples
- Mixed Level Modeling
- Module Summary

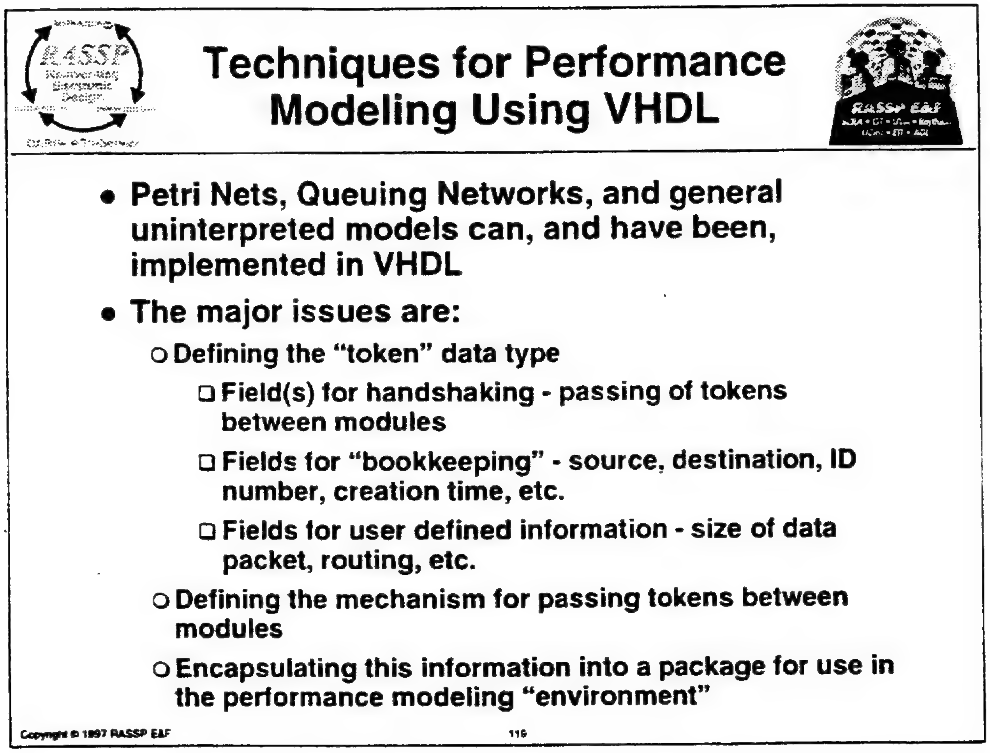
Module Outline

## Advantages of Using VHDL for Performance Modeling

- Adopted as a standard language and supported by many tools, vendors, and platforms
- Provides an expressive language with a built-in timing model, and full hierarchy and configurations which allows rapid development of highly flexible models of hardware
- Allows for easier consistency checks
- Provides a single language approach for system hardware modeling from concept to implementation
- Provides tight coupling to the lower levels of design
  - Mixed level modeling technique for model refinement can utilize off-the-shelf VHDL models for system components
  - High level performance model components written in VHDL can be used as starting point for fully behavioral and/or synthesizable VHDL models

As a hardware description language, VHDL has many desirable features for describing hardware already built-in such a a timing model, support for design hierarchy and configuration, etc. A general programming language such as C or C++ has none of these things.

A single language approach is beneficial because it means that hardware designers can work in VHDL to describe their components at all levels from the system level on down. Also, the system level VHDL models can be a starting point for fully behavioral or even synthesizable VHDL models of components.

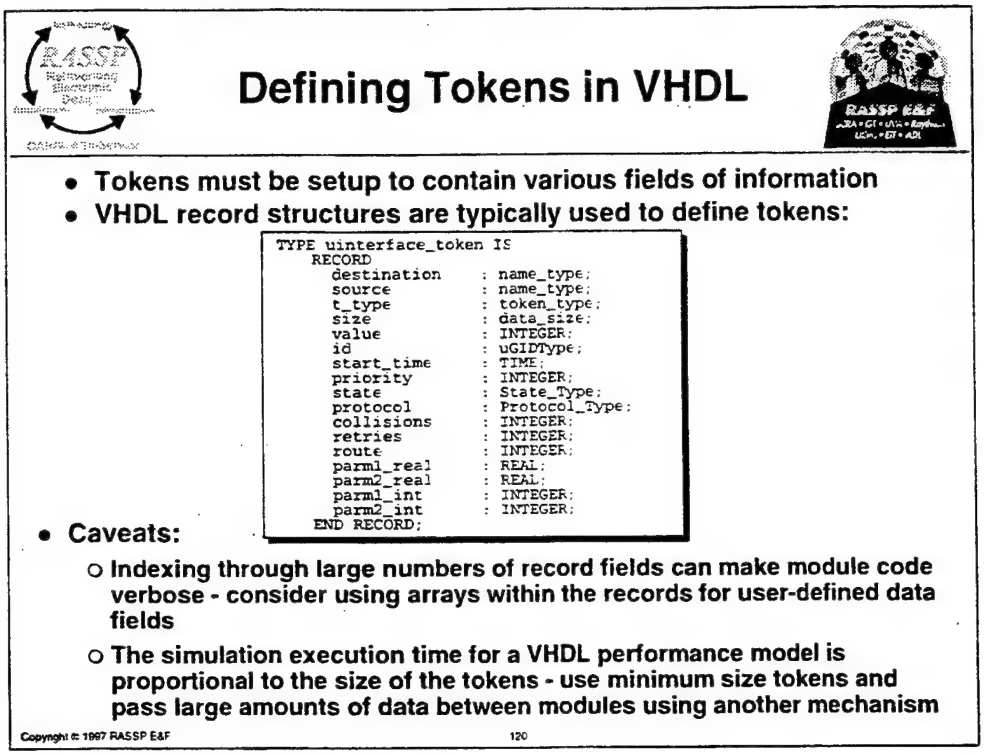


## Techniques for Performance Modeling Using VHDL

- **Petri Nets, Queuing Networks, and general uninterpreted models can, and have been, implemented in VHDL**
- **The major issues are:**
  - ○ Defining the "token" data type
    - ☐ Field(s) for handshaking - passing of tokens between modules
    - ☐ Fields for "bookkeeping" - source, destination, ID number, creation time, etc.
    - ☐ Fields for user defined information - size of data packet, routing, etc.
  - ○ Defining the mechanism for passing tokens between modules
  - ○ Encapsulating this information into a package for use in the performance modeling "environment"

Traditional performance modeling methods such as queuing models and Petri Nets, have been implemented in VHDL by UVa and others, as have more general uninterpreted performance modeling environments.

The major issues in this type of modeling effort in VHDL are discussed above.



## Defining Tokens in VHDL

- **Tokens must be setup to contain various fields of information**
- **VHDL record structures are typically used to define tokens:**

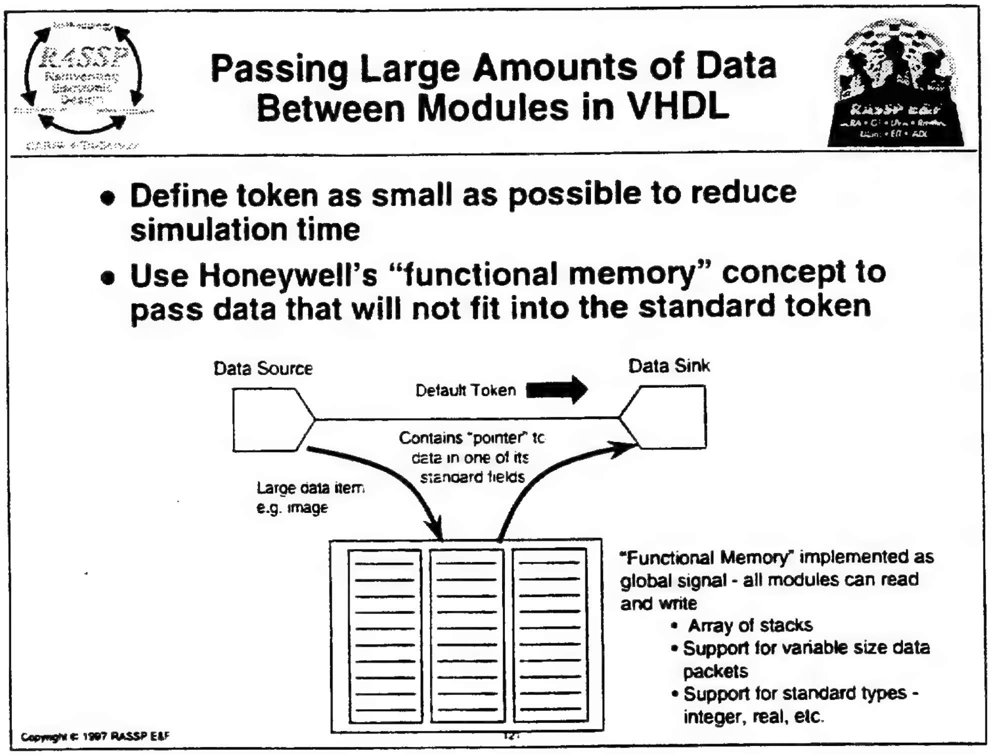
```
TYPE uinterface_token IS
    RECORD
      destination     : name_type;
      source          : name_type;
      t_type          : token_type;
      size            : data_size;
      value           : INTEGER;
      id              : uGIDType;
      start_time      : TIME;
      priority        : INTEGER;
      state           : State_Type;
      protocol        : Protocol_Type;
      collisions      : INTEGER;
      retries         : INTEGER;
      route           : INTEGER;
      parm1_real      : REAL;
      parm2_real      : REAL;
      parm1_int       : INTEGER;
      parm2_int       : INTEGER;
    END RECORD;
```

- **Caveats:**
  - ○ **Indexing through large numbers of record fields can make module code verbose - consider using arrays within the records for user-defined data fields**
  - ○ **The simulation execution time for a VHDL performance model is proportional to the size of the tokens - use minimum size tokens and pass large amounts of data between modules using another mechanism**

This slide includes the source code (somewhat modified) for the generic interface token developed by Honeywell Technology Center as an example.

Tokens in VHDL are probably best described as records. However, if large numbers of user defined fields are to be includes, it is sometimes better to define those as arrays within the record structure. This allows the code that accesses the user defined fields to do so with loops and to index them easily (e.g., token.user_array(value_one) ).

Another issue to consider is that it has become apparent that the size of the token has a great influence on the simulation time of the model, especially if a bus resolution function is used to pass tokens between modules.



## Passing Large Amounts of Data Between Modules in VHDL

- **Define token as small as possible to reduce simulation time**
- **Use Honeywell's "functional memory" concept to pass data that will not fit into the standard token**

Data Source

Default Token

Data Sink

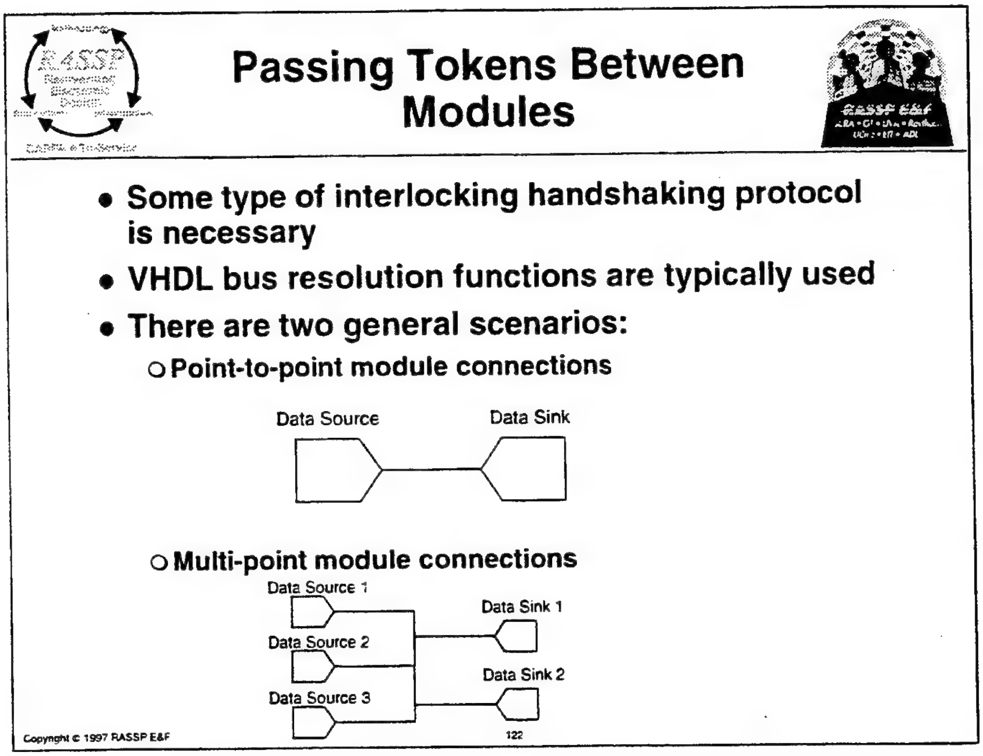
Contains "pointer" to data in one of its standard fields

Large data item e.g. image

"Functional Memory" implemented as global signal - all modules can read and write
- Array of stacks
- Support for variable size data packets
- Support for standard types - integer, real, etc.

The problem with passing large amounts of data in a token is that large tokens slow down the VHDL simulation greatly. Also, if only one token signal in a given model needs to carry a large amount of information, all tokens will be large (because they all have to be the same size) which is a waste of simulation speed and memory.

A solution developed by Honeywell as part of their PML (to be presented later) is to have a global signal, declared in a package and visible to all architectures, that can be used as a storage space to pass large amounts of data. Modules that want to pass data write it into this "functional memory" which is implemented as an array of stacks supporting generic types like integers and reals, and pass pointers to the information to other modules in one of the standard token fields. These other modules can then read the information out of the functional memory as required.



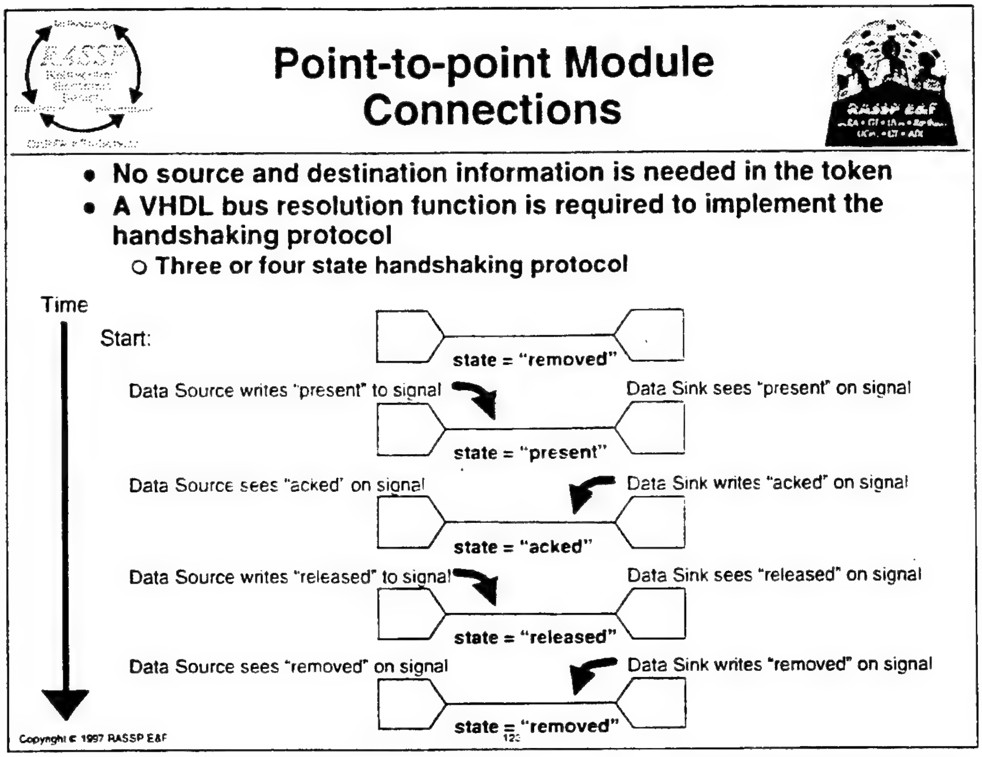
# Passing Tokens Between Modules

- Some type of interlocking handshaking protocol is necessary
- VHDL bus resolution functions are typically used
- There are two general scenarios:
  - ○ Point-to-point module connections
  - ○ Multi-point module connections

Data Source
Data Sink
Data Source 1
Data Sink 1
Data Source 2
Data Sink 2
Data Source 3

Some type if interlocking mechanism to pass tokens from one module to another is necessary. VHDL bus resolution functions are typically used, both in the point-to-point and multiple driver/reader case, because the token signal is bi-directional. That is, the data source has to be able to drive the new token onto the signal and the data destination has to be able to drive the acknowledgement onto the signal. The two sources require a resolution function.

An alternative (used in the ATL models and in the latest version of ADEPT) is to have unidirectional signals, one from source to destination to place the initial token, and another from the destination to the source to acknowledge the token.



## Point-to-point Module Connections

- **No source and destination information is needed in the token**
- **A VHDL bus resolution function is required to implement the handshaking protocol**
  - ○ **Three or four state handshaking protocol**

Time

Start:

**state = "removed"**

Data Source writes "present" to signal

Data Sink sees "present" on signal

**state = "present"**

Data Source sees "acked" on signal

Data Sink writes "acked" on signal

**state = "acked"**

Data Source writes "released" to signal

Data Sink sees "released" on signal

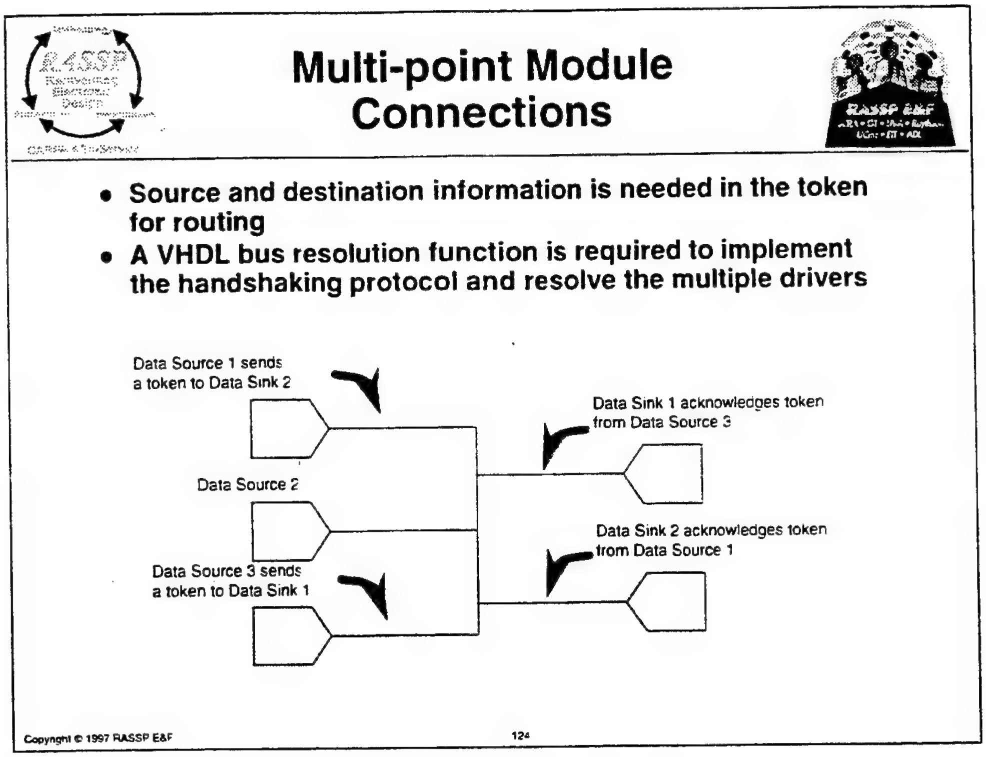
**state = "released"**

Data Source sees "removed" on signal

Data Sink writes "removed" on signal

**state = "removed"**

This is an example of how a four state, point-to-point token passing protocol works and why it need a resolution function (taken from ADEPT).



# Multi-point Module Connections

- **Source and destination information is needed in the token for routing**
- **A VHDL bus resolution function is required to implement the handshaking protocol and resolve the multiple drivers**

Data Source 1 sends a token to Data Sink 2

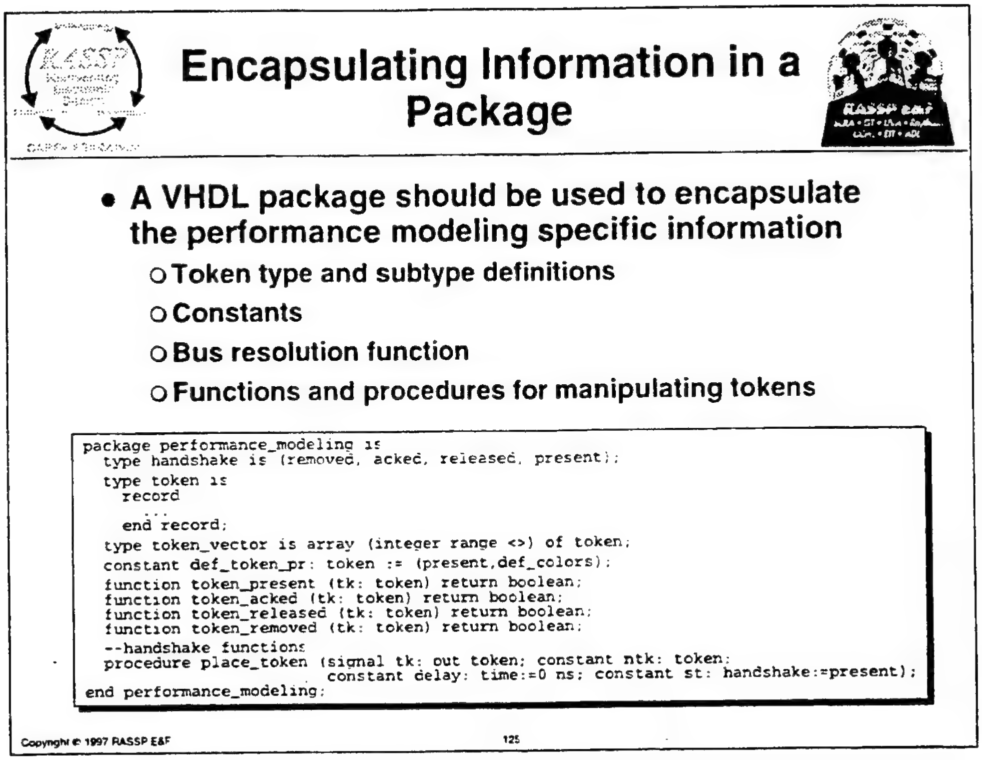
Data Source 2

Data Source 3 sends a token to Data Sink 1

Data Sink 1 acknowledges token from Data Source 3

Data Sink 2 acknowledges token from Data Source 1

This is a multipoint communications protocol. Why a bus resolution function is needed here is self-evident. This is the token passing protocol used in the Honeywell PML, Cosmos.

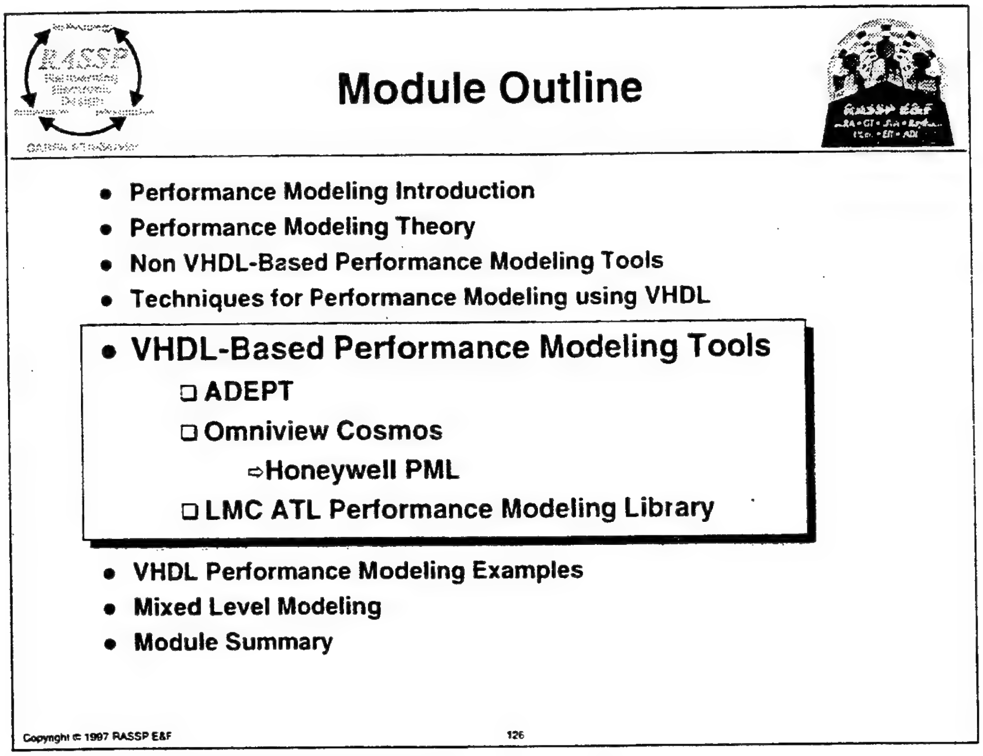


# Encapsulating Information in a Package

- **A VHDL package should be used to encapsulate the performance modeling specific information**
  - ○ Token type and subtype definitions
  - ○ Constants
  - ○ Bus resolution function
  - ○ Functions and procedures for manipulating tokens

```
package performance_modeling is
  type handshake is (removed, acked, released, present);
  type token is
    record
      ...
    end record;
  type token_vector is array (integer range <>) of token;
  constant def_token_pr: token := (present,def_colors);
  function token_present (tk: token) return boolean;
  function token_acked (tk: token) return boolean;
  function token_released (tk: token) return boolean;
  function token_removed (tk: token) return boolean;
  --handshake functions
  procedure place_token (signal tk: out token; constant ntk: token;
                         constant delay: time:=0 ns; constant st: handshake:=present);
end performance_modeling;
```

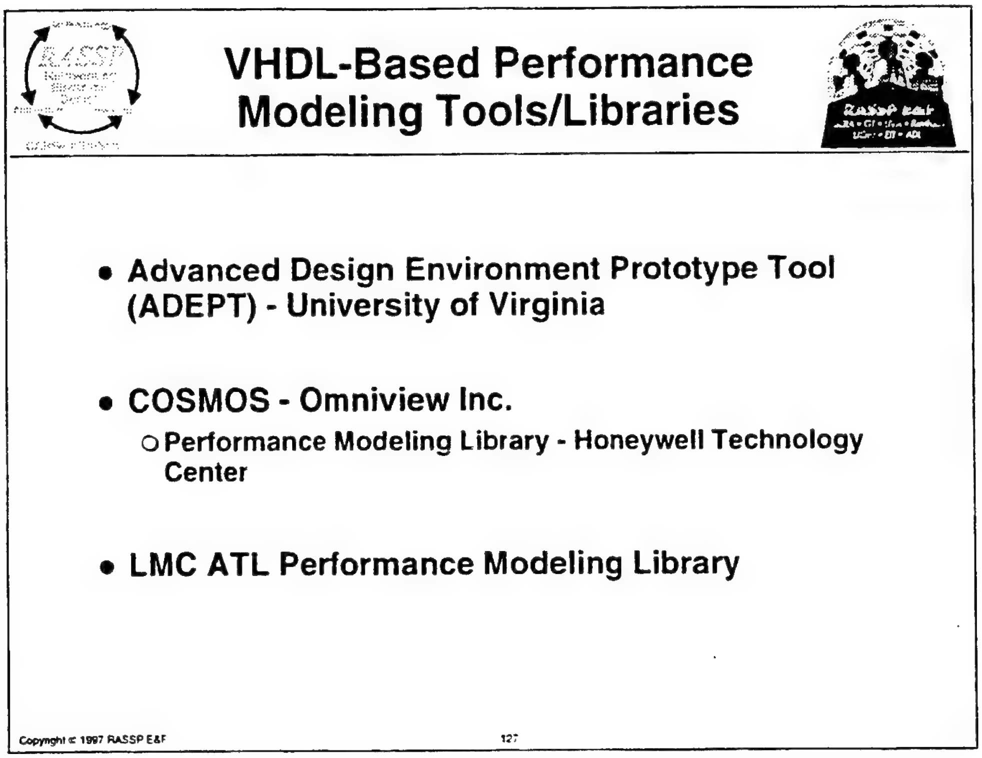
Finally, once all of the information necessary to do performance modeling is defined (types, functions, procedures), it should be encapsulated into a package that can be made visible to any performance modeling component that needs it.



## Module Outline

- Performance Modeling Introduction
- Performance Modeling Theory
- Non VHDL-Based Performance Modeling Tools
- Techniques for Performance Modeling using VHDL
- **VHDL-Based Performance Modeling Tools**
  - ❑ ADEPT
  - ❑ Omniview Cosmos
    - ⇨Honeywell PML
  - ❑ LMC ATL Performance Modeling Library
- VHDL Performance Modeling Examples
- Mixed Level Modeling
- Module Summary

Module Outline



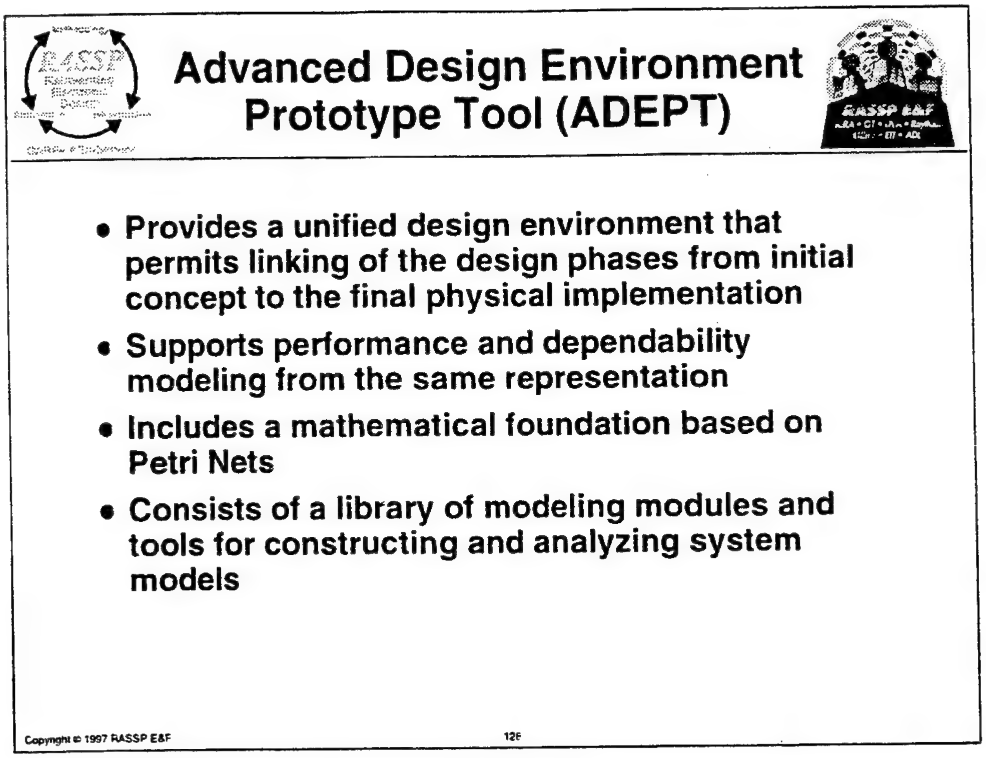
## VHDL-Based Performance Modeling Tools/Libraries

- **Advanced Design Environment Prototype Tool (ADEPT) - University of Virginia**
- **COSMOS - Omniview Inc.**
  - ○ **Performance Modeling Library - Honeywell Technology Center**
- **LMC ATL Performance Modeling Library**

UVa's ADEPT system is a set of library elements and a set of tools for constructing VHDL performance models.

Omniview's Cosmos product is a set of tools for constructing and analyzing the results of, VHDL performance models. It includes a performance modeling library based on the Performance Modeling Library developed by Honeywell Technology Center.

The Lockheed Martin, Advanced Technology Laboratory has developed a small library of VHDL performance modeling elements, specifically targeted at modeling Mercury Race Multicomputers, and a few tools for analyzing their results.



## Advanced Design Environment Prototype Tool (ADEPT)

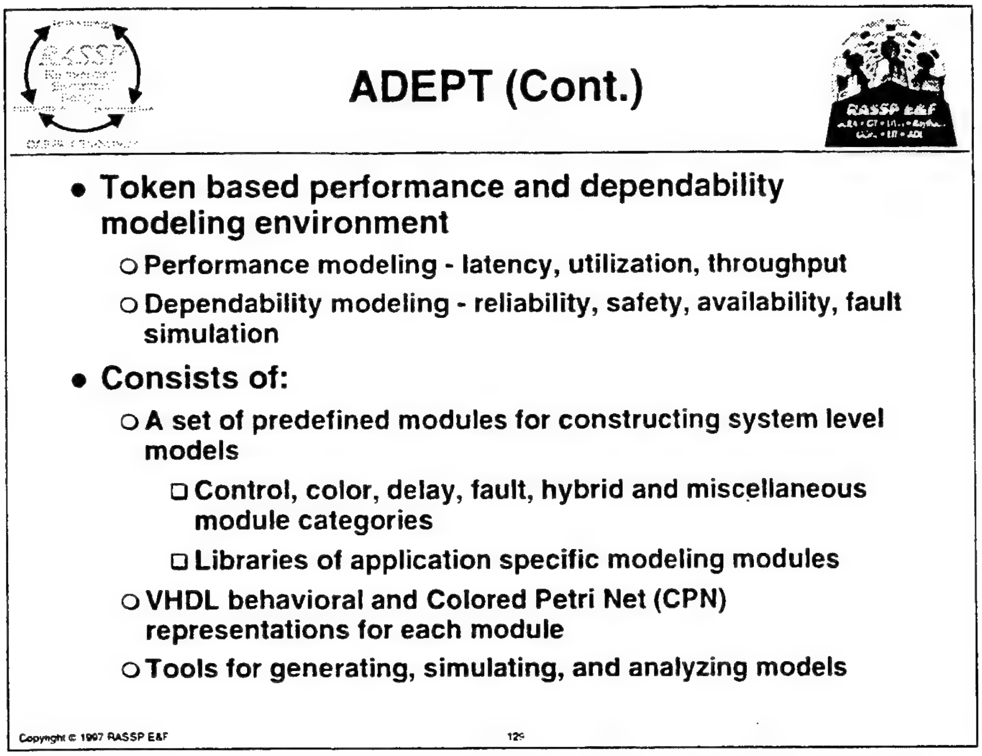
- Provides a unified design environment that permits linking of the design phases from initial concept to the final physical implementation
- Supports performance and dependability modeling from the same representation
- Includes a mathematical foundation based on Petri Nets
- Consists of a library of modeling modules and tools for constructing and analyzing system models

The Advanced Prototype Design Environment from UVa is a general VHDL-based uninterpreted modeling environment that also includes a Petri Net foundation (as will be explained). It consists of a library of modules for constructing system-level performance and Dependability models, and a set of tools for constructing and analyzing those models.

More information, including complete documentation and source code for ADEPT can be found on the UVa RASSP web page:

http://csis.ee.virginia/~rassp

under the Publications and Tools sections. This includes some more detailed examples of performance, dependability, and mixed level modeling using ADEPT.



## ADEPT (Cont.)

- Token based performance and dependability modeling environment
    - Performance modeling - latency, utilization, throughput
    - Dependability modeling - reliability, safety, availability, fault simulation
- Consists of:
    - A set of predefined modules for constructing system level models
        - Control, color, delay, fault, hybrid and miscellaneous module categories
        - Libraries of application specific modeling modules
    - VHDL behavioral and Colored Petri Net (CPN) representations for each module
    - Tools for generating, simulating, and analyzing models

ADEPT's strengths consist of:

- the inclusion of a mathematical foundation which makes analytical analysis of ADEPT models possible,
- the capability to perform performance and reliability modeling from the same ADEPT model without modification,
- the inclusion of a library of elements with which interfaces to behavioral models can be easily constructed for mixed level modeling, and
- the ability of the user to easily extend the ADEPT libraries.

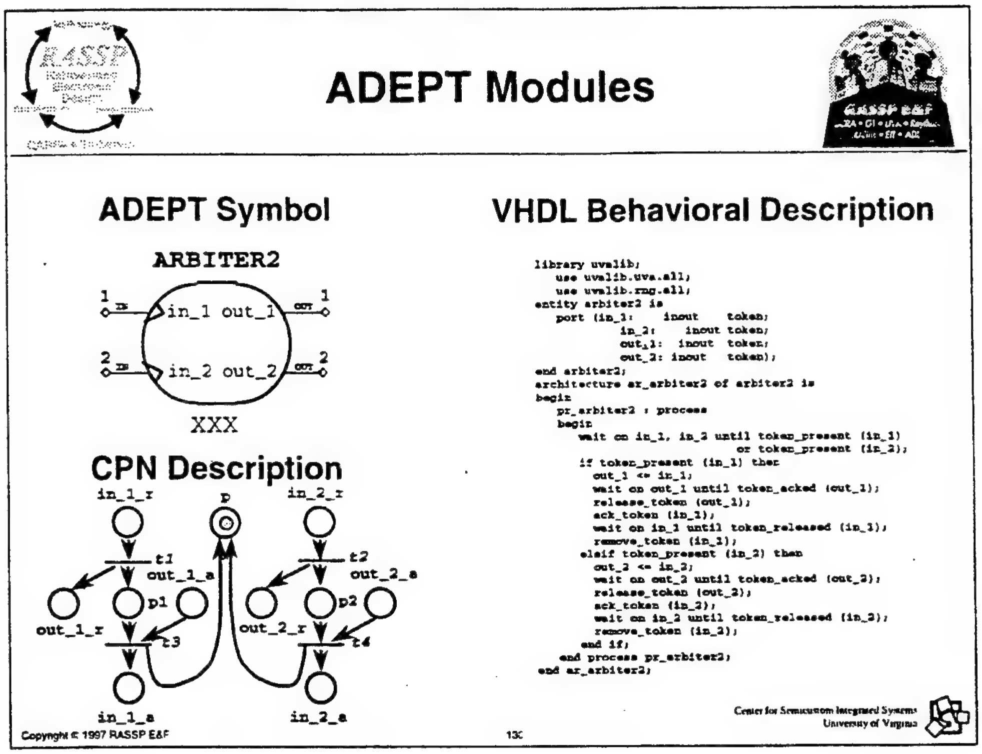
ADEPT's weaknesses include:

- the fact that the low level nature of the ADEPT modules sometimes makes model construction difficult and time consuming[1], and
- the fact that because its VHDL based, simulation of ADEPT models can take a long time[2].

Notes:

1) This is being alleviated somewhat by the addition of libraries of more complex modules, although these modules often lack the Petri Net representation.

2) This is being addressed by an effort to simplify and speedup the simulation of ADEPT models.



## ADEPT Modules

### ADEPT Symbol

### CPN Description

### VHDL Behavioral Description

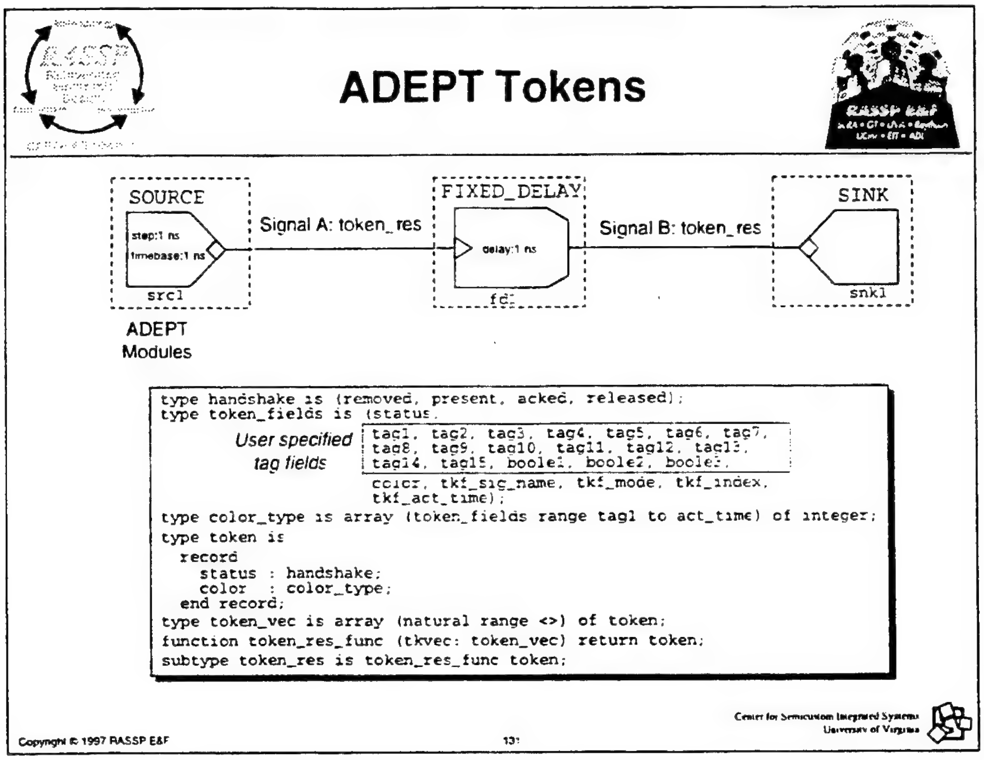
```
library uvalib;
   use uvalib.uva.all;
   use uvalib.rng.all;
entity arbiter2 is
   port (in_1:    inout    token;
         in_2:      inout token;
         out_1:   inout  token;
         out_2: inout    token);
end arbiter2;
architecture ar_arbiter2 of arbiter2 is
begin
   pr_arbiter2 : process
   begin
      wait on in_1, in_2 until token_present (in_1)
                          or token_present (in_2);
      if token_present (in_1) then
        out_1 <= in_1;
        wait on out_1 until token_acked (out_1);
        release_token (out_1);
        ack_token (in_1);
        wait on in_1 until token_released (in_1);
        remove_token (in_1);
      elsif token_present (in_2) then
        out_2 <= in_2;
        wait on out_2 until token_acked (out_2);
        release_token (out_2);
        ack_token (in_2);
        wait on in_2 until token_released (in_2);
        remove_token (in_2);
      end if;
   end process pr_arbiter2;
end ar_arbiter2;
```

Center for Semicustom Integrated Systems
University of Virginia

Copyright © 1997 RASSP E&F

130

This figure shows the ADEPT symbol for an arbiter module - a module that serializes two tokens that arrive simultaneously on its inputs - its corresponding VHDL behavioral description, and its corresponding Colored Petri Net description. All of the ADEPT modules have a symbol and VHDL behavioral description that can be used for simulation. The ADEPT primitive modules - those in the Control, Color, Delay, Fault, Miscellaneous, and Hybrid categories - have colored Petri Net descriptions.



# ADEPT Tokens

SOURCE
step:1 ns
timebase:1 ns
src1
Signal A: token_res
FIXED_DELAY
delay:1 ns
fd1
Signal B: token_res
SINK
snk1
ADEPT Modules

*User specified tag fields*

```
type handshake is (removed, present, acked, released);
type token_fields is (status,
                      tag1, tag2, tag3, tag4, tag5, tag6, tag7,
                      tag8, tag9, tag10, tag11, tag12, tag13,
                      tag14, tag15, boole1, boole2, boole3,
                      color, tkf_sig_name, tkf_mode, tkf_index,
                      tkf_act_time);
type color_type is array (token_fields range tag1 to act_time) of integer;
type token is
  record
    status : handshake;
    color  : color_type;
  end record;
type token_vec is array (natural range <>) of token;
function token_res_func (tkvec: token_vec) return token;
subtype token_res is token_res_func token;
```

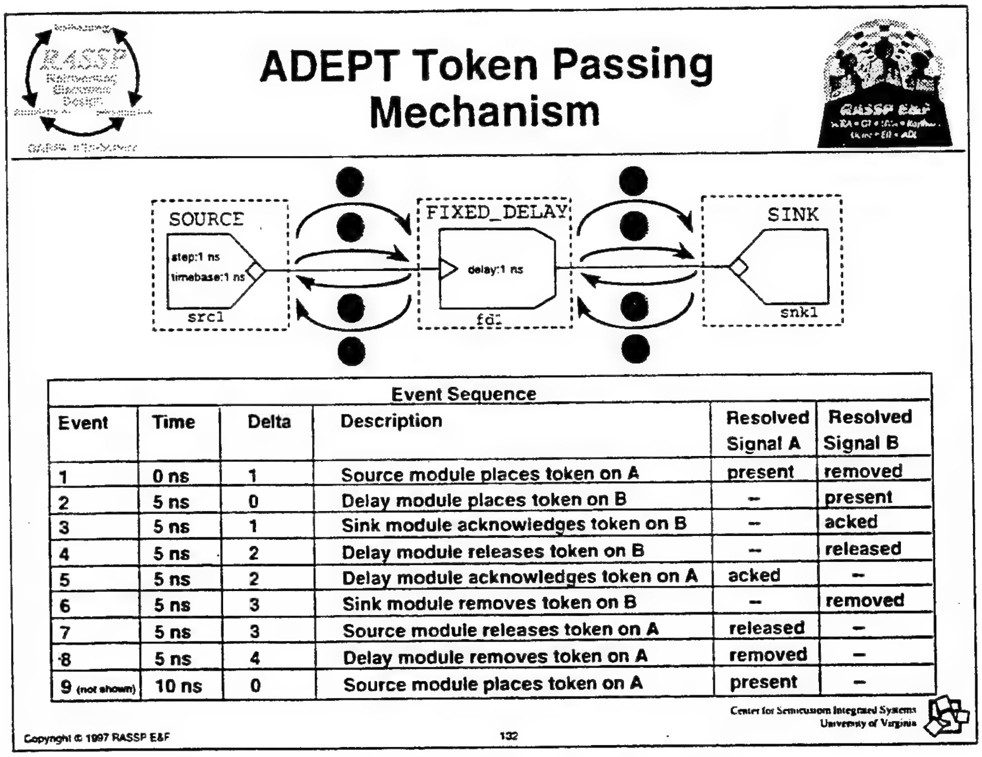
131

Center for Semicustom Integrated Systems
University of Virginia

ADEPT modules are connected via VHDL signals. These signals carry the tokens between the modules. The ADEPT tokens are implemented in VHDL as a record structure with two fields, a status field that is used to implement the 4 state handshaking, and a color field which is an array of integers used to hold user-defined information.

A VHDL bus resolution function, called token_res_function, is used to implement the point-to-point token passing mechanism as described earlier.

The point-to-point token mechanism uses a 4 state, fully-interlocked protocol. The states (enumerated in the handshake type) are "present," "ack(nowledg)ed," "released," and "removed."


SOURCE
step:1 ns
timebase:1 ns
src1
FIXED_DELAY
delay:1 ns
fd1
SINK
snk1

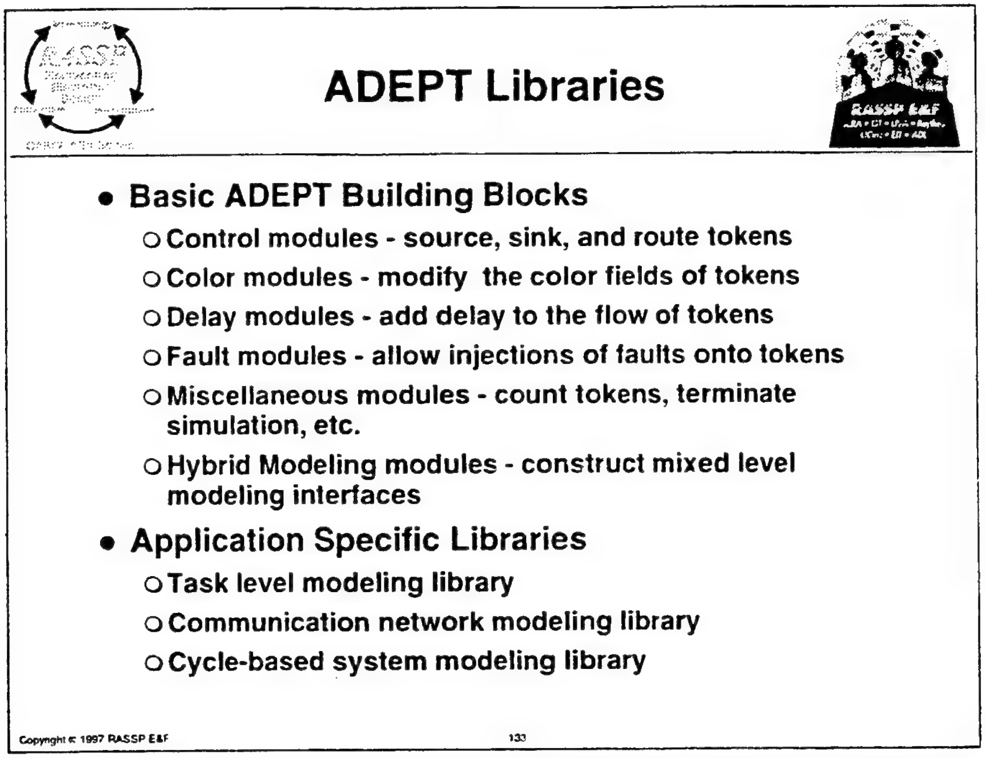
# ADEPT Token Passing Mechanism

**Event Sequence**

| Event | Time | Delta | Description | Resolved Signal A | Resolved Signal B |
|---|---|---|---|---|---|
| 1 | 0 ns | 1 | Source module places token on A | present | removed |
| 2 | 5 ns | 0 | Delay module places token on B | – | present |
| 3 | 5 ns | 1 | Sink module acknowledges token on B | – | acked |
| 4 | 5 ns | 2 | Delay module releases token on B | – | released |
| 5 | 5 ns | 2 | Delay module acknowledges token on A | acked | – |
| 6 | 5 ns | 3 | Sink module removes token on B | – | removed |
| 7 | 5 ns | 3 | Source module releases token on A | released | – |
| 8 | 5 ns | 4 | Delay module removes token on A | removed | – |
| 9 (not shown) | 10 ns | 0 | Source module places token on A | present | – |

Center for Semicustom Integrated Systems
University of Virginia

This is a detailed description of the ADEPT token passing protocol using a simple source/delay/sink model. Note that the only time that actually passes in the model is that taken up by the delay module - the token handshaking takes place in VHDL delta cycles with no time delay. In general, only delay module in ADEPT have actual time delays associated with them. All other modules use only delta delay. This fact can sometimes cause problems (delta cycle races) in constructing an ADEPT model.



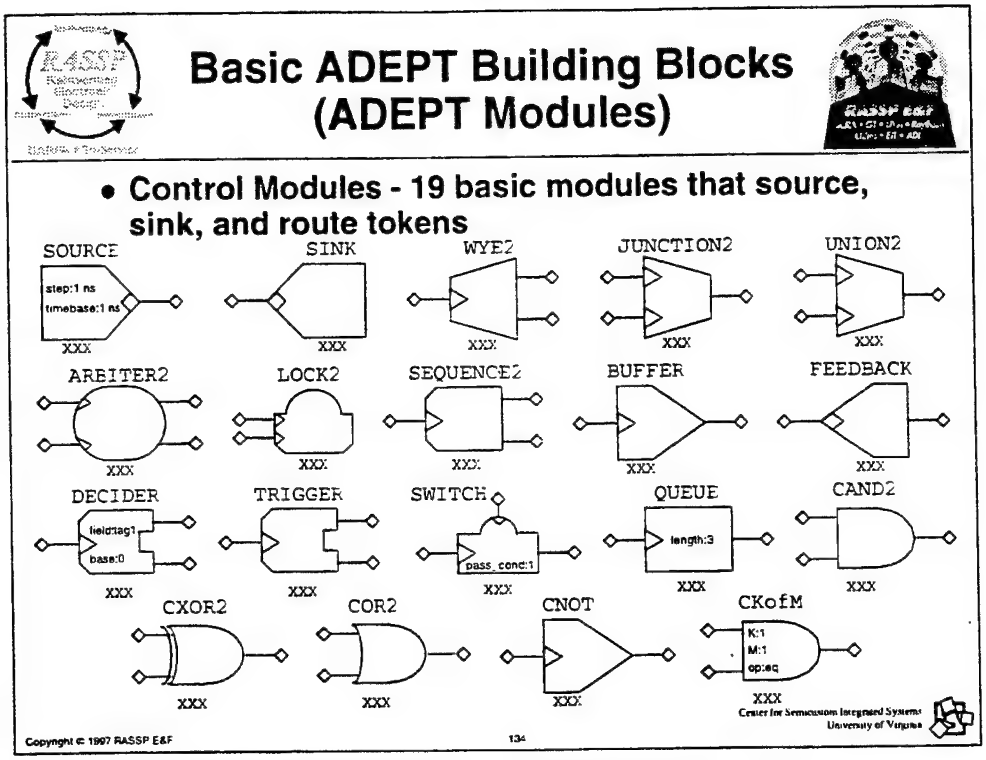
## ADEPT Libraries

- Basic ADEPT Building Blocks
  - ○ Control modules - source, sink, and route tokens
  - ○ Color modules - modify the color fields of tokens
  - ○ Delay modules - add delay to the flow of tokens
  - ○ Fault modules - allow injections of faults onto tokens
  - ○ Miscellaneous modules - count tokens, terminate simulation, etc.
  - ○ Hybrid Modeling modules - construct mixed level modeling interfaces
- Application Specific Libraries
  - ○ Task level modeling library
  - ○ Communication network modeling library
  - ○ Cycle-based system modeling library

There are six categories of basic ADEPT building blocks out of which general system models can be constructed. As stated previously, these module have both a VHDL behavioral description and the Colored Petri Net description.

Because of the difficulty with which users have been constructing complex models out of the basic building blocks, libraries of more complex constructs and modeling modules have been developed. The elements in these libraries, which are targeted towards modeling systems in certain application areas, have only the VHDL behavioral description for simulation.

See [ADEPT_LR96] for more details on all of the ADEPT modules.



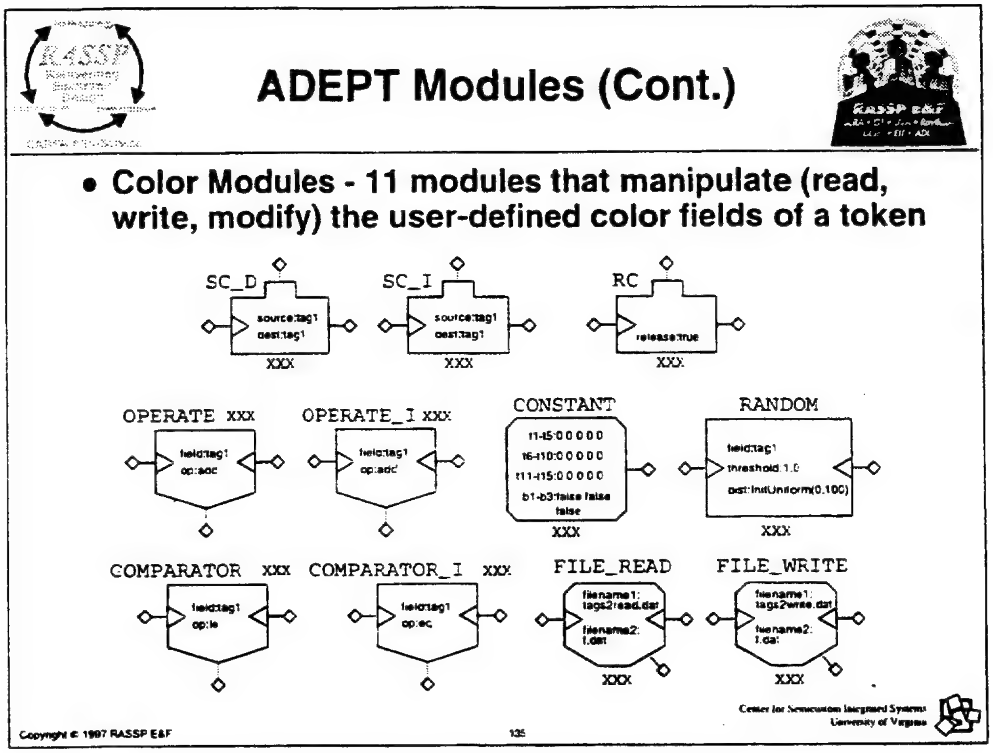
## Basic ADEPT Building Blocks (ADEPT Modules)

- **Control Modules - 19 basic modules that source, sink, and route tokens**

Center for Semiconductor Integrated Systems
University of Virginia

There are 19 modules in the Control category. These modules include the source and sink module for creating and destroying tokens, the wye, junction and union modules for fanning in and fanning out tokens, the buffer and feedback modules for buffering parts of the a system model from others, queue modules, for storing tokens, and other modules for routing tokens within a model.

There are also the "C" modules, like the CNOT and CXOR, that manipulate so called "control," or independent tokens. In ADEPT, the tokens that are passed between modules using the 4 state interlocked protocol, are called "data" or dependent tokens. Independent or "control" tokens are tokens which have one source, but no real sinks. Then can take on only two of the 4 states in the protocol, present and released. They are generally used to carry routing and control information. For example, the output from the queue module which tells if the queue is full or not, and the inputs to the decider and switch module which determine if, and which output is active, are "control" tokens. See [ADEPT_UM96] for more details.



## ADEPT Modules (Cont.)

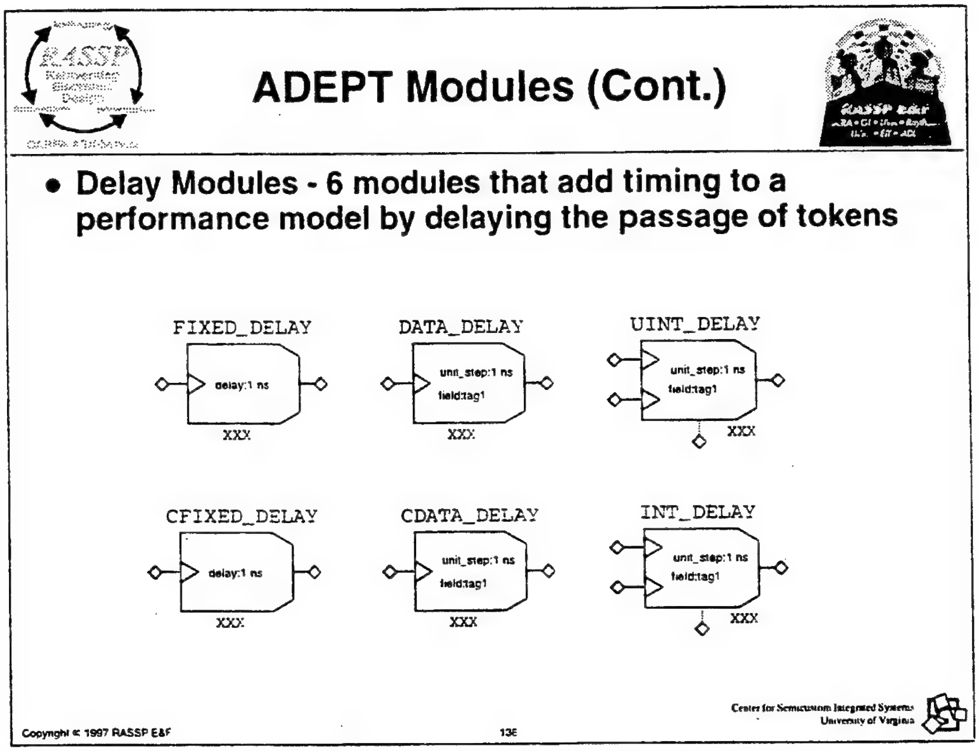
- **Color Modules - 11 modules that manipulate (read, write, modify) the user-defined color fields of a token**

SC_D
source:tag1
dest:tag1
XXX
SC_I
source:tag1
dest:tag1
XXX
RC
release:true
XXX
OPERATE XXX
field:tag1
op:add
OPERATE_I XXX
field:tag1
op:add
CONSTANT
t1-t5:0 0 0 0 0
t6-t10:0 0 0 0 0
t11-t15:0 0 0 0 0
b1-b3:false false false
XXX
RANDOM
field:tag1
threshold:1.0
dist:InitUniform(0.100)
XXX
COMPARATOR XXX
field:tag1
op:le
COMPARATOR_I XXX
field:tag1
op:eq
FILE_READ
filename1: tags2read.dat
filename2: f.dat
XXX
FILE_WRITE
filename1: tags2write.dat
filename2: f.dat
XXX

135

Center for Semicustom Integrated Systems
University of Virginia

The color modules are used to access the user-defined (color fields) of the tokens. The set color (SC_D, SC_I) modules set values on tokens passing through them, and does the file_read module the read color (RC) module and the file_write module read color fields and write them onto other tokens or a file. The operator and comparator modules allow arithmetic and logical operations with token color fields, and the random module puts a random value on a color field.

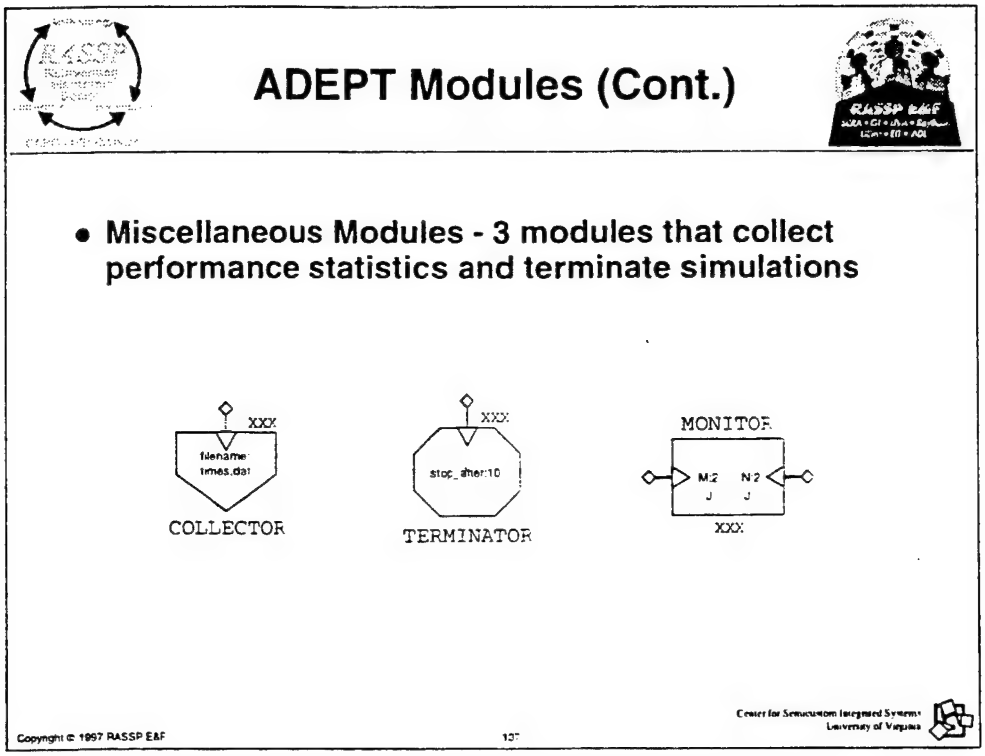


## ADEPT Modules (Cont.)

- **Delay Modules - 6 modules that add timing to a performance model by delaying the passage of tokens**

Center for Semicustom Integrated Systems
University of Virginia

As stated previously, the delay modules are the only modules in the basic ADEPT set that have simulation time associated with them. There are fixed and data dependent delays for both "data" and "control" type tokens and more complex delay modules for modeling synchronization type events.

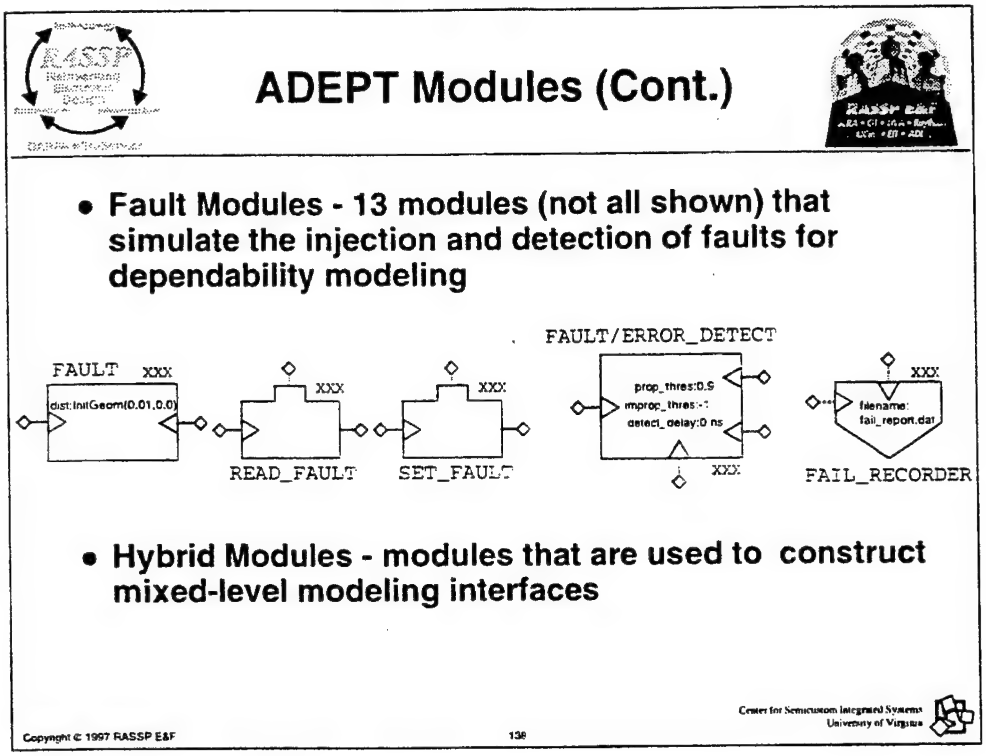


## ADEPT Modules (Cont.)

- **Miscellaneous Modules - 3 modules that collect performance statistics and terminate simulations**

Center for Semicustom Integrated Systems
University of Virginia

The miscellaneous module category includes the collector, which writes the time that a token passes a certain point in the model to a file, the terminator module, which can stop a simulation after a chosen number of tokens have gone past a specific point, and the monitor module, which writes latency and utilization data out to a file for post-processing by the ADEPT tools.



## ADEPT Modules (Cont.)

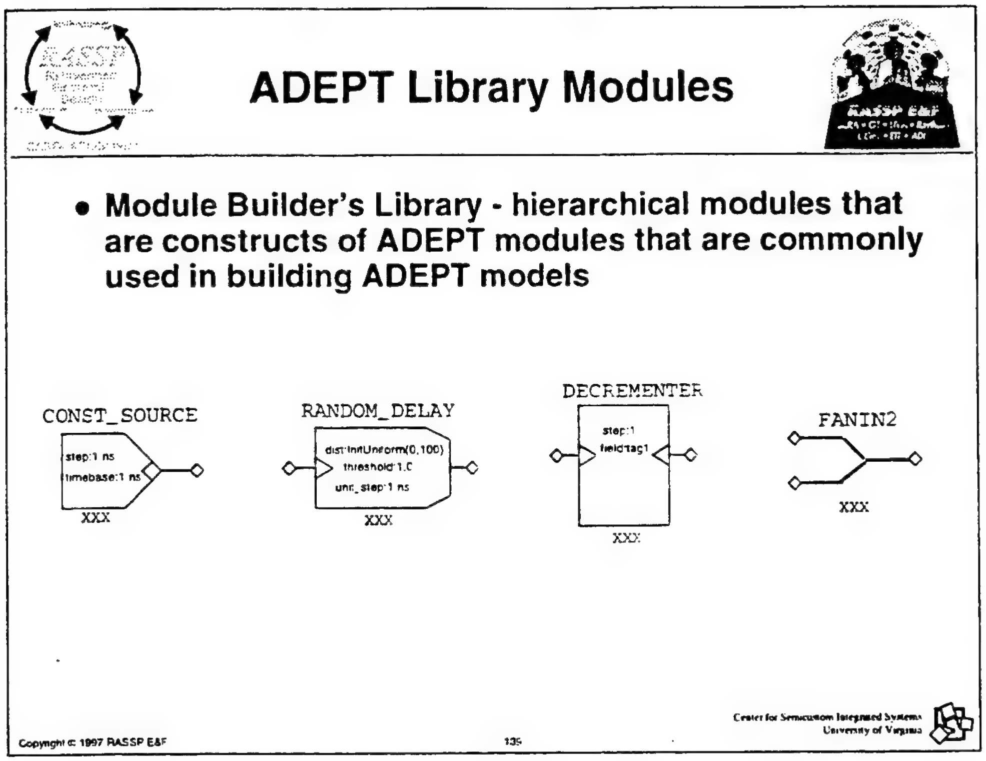
- **Fault Modules - 13 modules (not all shown) that simulate the injection and detection of faults for dependability modeling**

FAULT
xxx
dist:InitGeom(0.01,0.0)
XXX
READ_FAULT
XXX
SET_FAULT
FAULT/ERROR_DETECT
prop_thres:0.9
improp_thres:-1
detect_delay:0 ns
XXX
XXX
filename:
fail_report.dat
FAIL_RECORDER

- **Hybrid Modules - modules that are used to construct mixed-level modeling interfaces**

Center for Semicustom Integrated Systems
University of Virginia

The fault modules allow the insertion and detection of faults into an ADEPT model for reliability analysis.

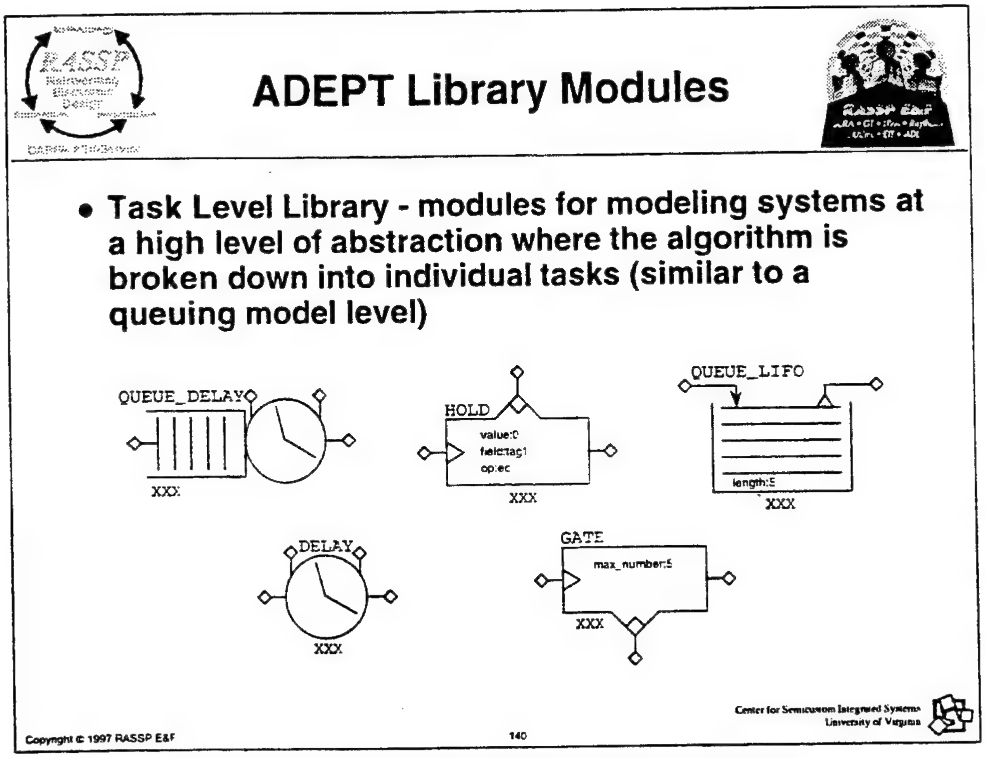


## ADEPT Library Modules

- **Module Builder's Library - hierarchical modules that are constructs of ADEPT modules that are commonly used in building ADEPT models**

Center for Semicustom Integrated Systems
University of Virginia

The Module Builders Library is a library of constructs commonly used in constructing ADEPT models. For example, the random delay module delays a token according to a random number. It is a hierarchical module built up mainly from a Random module and a Data Delay module. The Decrementer module will decrement the value on a token tag by a set amount. It is built up from a Read Color, Operator, and Set Color module.



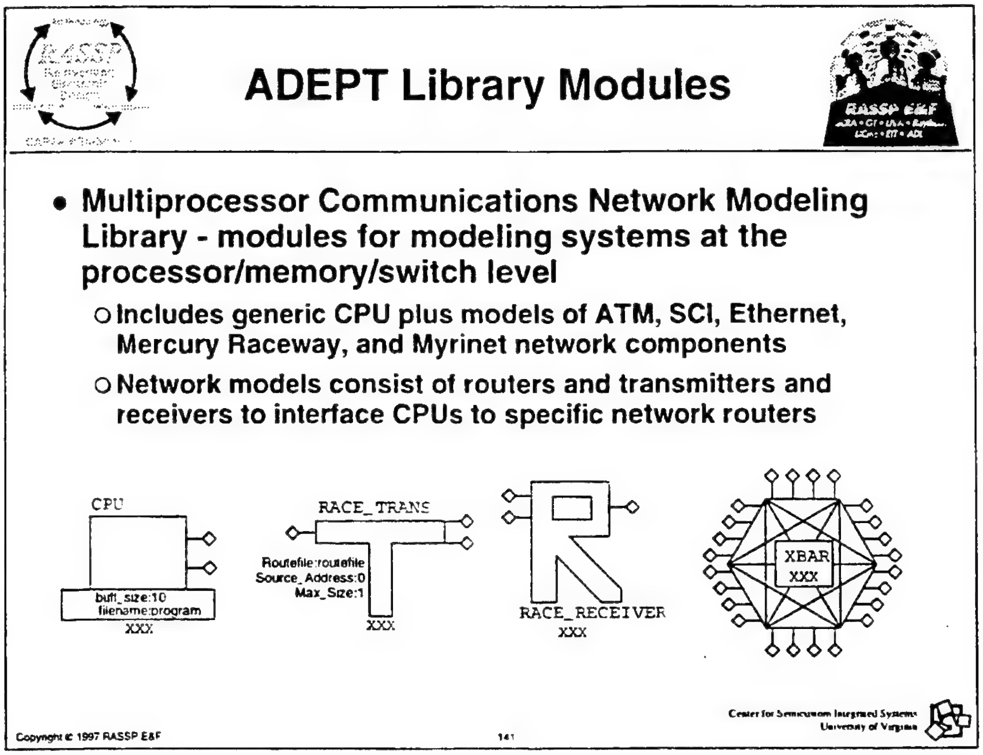
## ADEPT Library Modules

- **Task Level Library - modules for modeling systems at a high level of abstraction where the algorithm is broken down into individual tasks (similar to a queuing model level)**

QUEUE_DELAY
XXX
HOLD
value:0
field:tag1
op:ec
XXX
QUEUE_LIFO
length:5
XXX
DELAY
XXX
GATE
max_number:5
XXX

Center for Semicustom Integrated Systems
University of Virginia

The Task Level Library is intended to allow users to build high level models of various application areas. The elements in this library consist of various Server module, various type of queue, like FIFO, LIFO, and Priority, and special routing modules like the gate and hold. The modules in this library were modeled, to some extent, on the types of modules available in the Extend tool from Imagine That Inc.

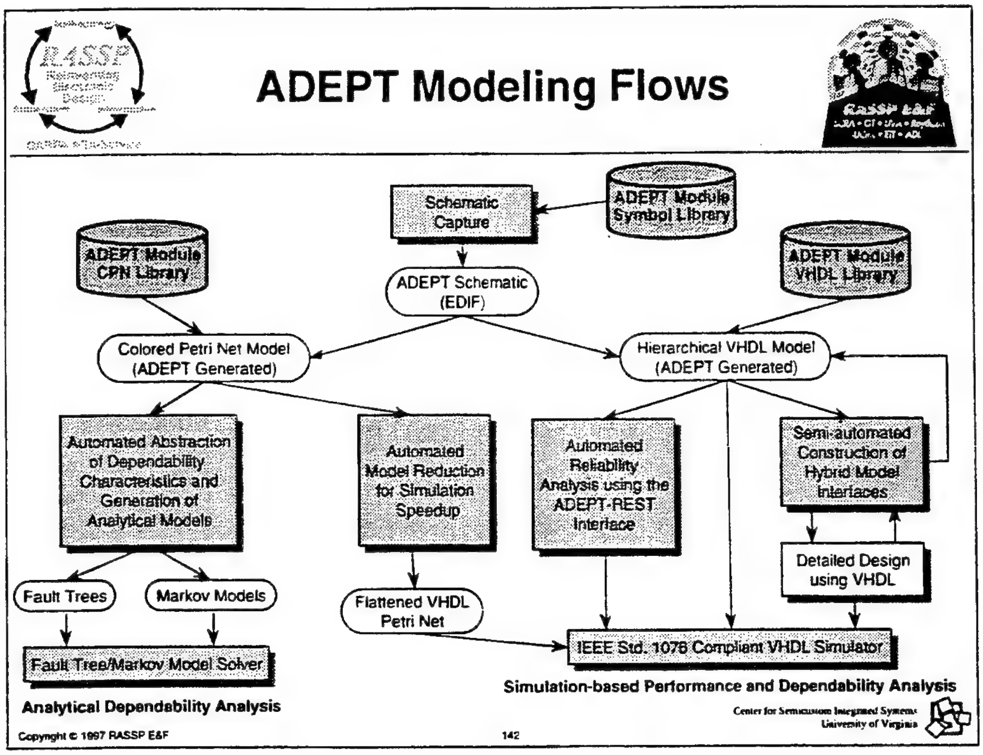


## ADEPT Library Modules

- **Multiprocessor Communications Network Modeling Library - modules for modeling systems at the processor/memory/switch level**
  - ○ **Includes generic CPU plus models of ATM, SCI, Ethernet, Mercury Raceway, and Myrinet network components**
  - ○ **Network models consist of routers and transmitters and receivers to interface CPUs to specific network routers**

Center for Semicustom Integrated Systems
University of Virginia

The Multiprocessor Communication Network Modeling library was developed under the RASSP program to ease modeling of embedded multicomputer applications. It includes a generic CPU, much like the ATL CPU model to be discussed, and network modules to model Raceway, Myrinet, SCI, Ethernet, and ATM networks.



## ADEPT Modeling Flows

Schematic Capture

ADEPT Module Symbol Library

ADEPT Module CPN Library

ADEPT Schematic (EDIF)

ADEPT Module VHDL Library

Colored Petri Net Model (ADEPT Generated)

Hierarchical VHDL Model (ADEPT Generated)

Automated Abstraction of Dependability Characteristics and Generation of Analytical Models

Automated Model Reduction for Simulation Speedup

Automated Reliability Analysis using the ADEPT-REST Interface

Semi-automated Construction of Hybrid Model Interfaces

Detailed Design using VHDL

Fault Trees

Markov Models

Flattened VHDL Petri Net

IEEE Std. 1076 Compliant VHDL Simulator

Fault Tree/Markov Model Solver

**Simulation-based Performance and Dependability Analysis**

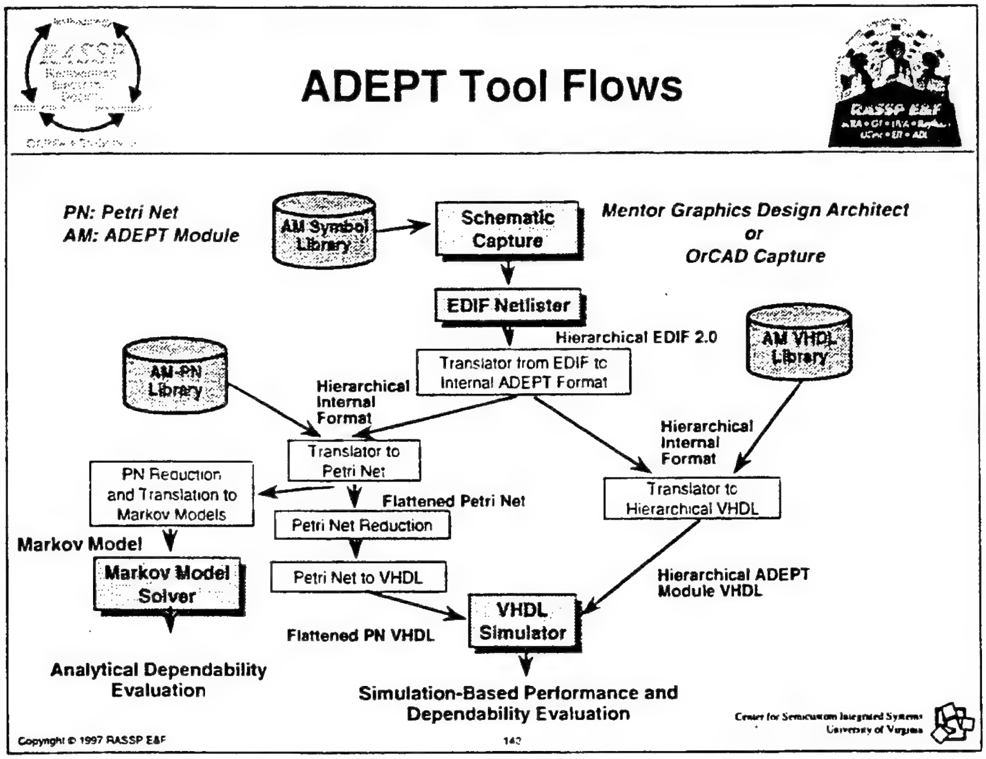
**Analytical Dependability Analysis**

Center for Semicustom Integrated Systems
University of Virginia

Copyright © 1997 RASSP E&F

142

This is a representation of the ADEPT modeling flows. Notice that there are two basic types of analysis, analytical (mainly for dependability modeling) and simulation-based (for both dependability and performance modeling). The boxes shown in blue are processes that are automated by tools developed for the ADEPT environment and the blue drums are ADEPT libraries of symbols, VHDL behavioral descriptions, and CPN descriptions.



# ADEPT Tool Flows

*PN: Petri Net*
*AM: ADEPT Module*

AM Symbol Library

**Schematic Capture**

*Mentor Graphics Design Architect*
*or*
*OrCAD Capture*

**EDIF Netlister**

**Hierarchical EDIF 2.0**

Translator from EDIF to Internal ADEPT Format

AM-PN Library

AM VHDL Library

**Hierarchical Internal Format**

Translator to Petri Net

**Hierarchical Internal Format**

Translator to Hierarchical VHDL

PN Reduction and Translation to Markov Models

**Flattened Petri Net**

Petri Net Reduction

**Markov Model**

**Markov Model Solver**

Petri Net to VHDL

**Hierarchical ADEPT Module VHDL**

**VHDL Simulator**

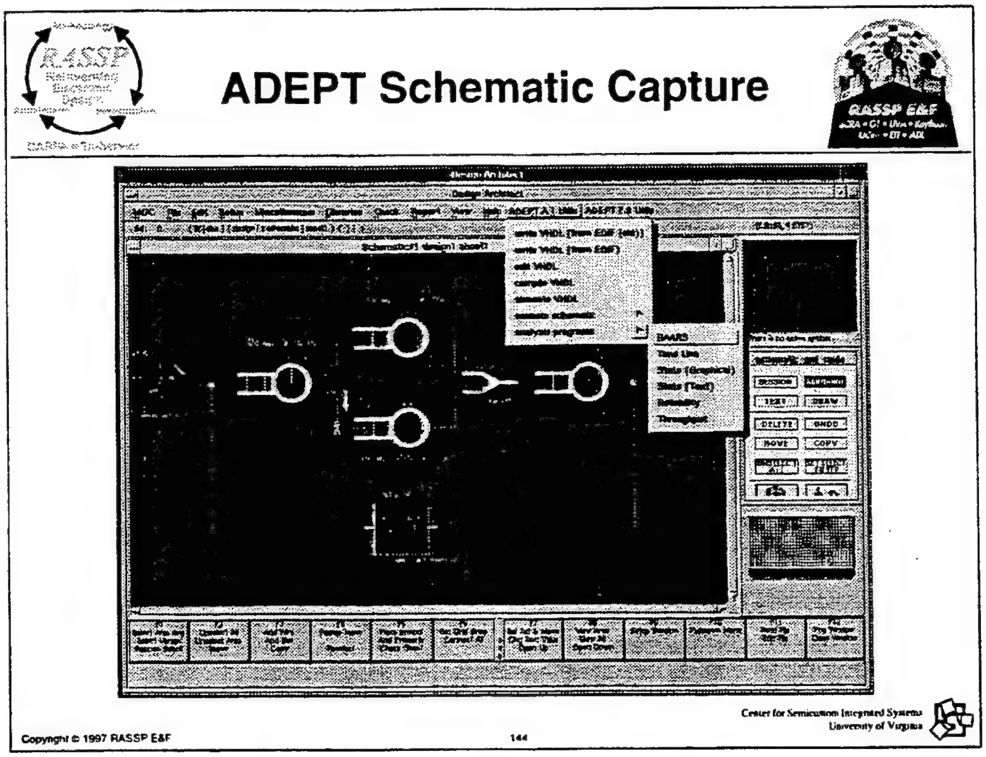
**Flattened PN VHDL**

**Analytical Dependability Evaluation**

**Simulation-Based Performance and Dependability Evaluation**

Center for Semicustom Integrated Systems
University of Virginia

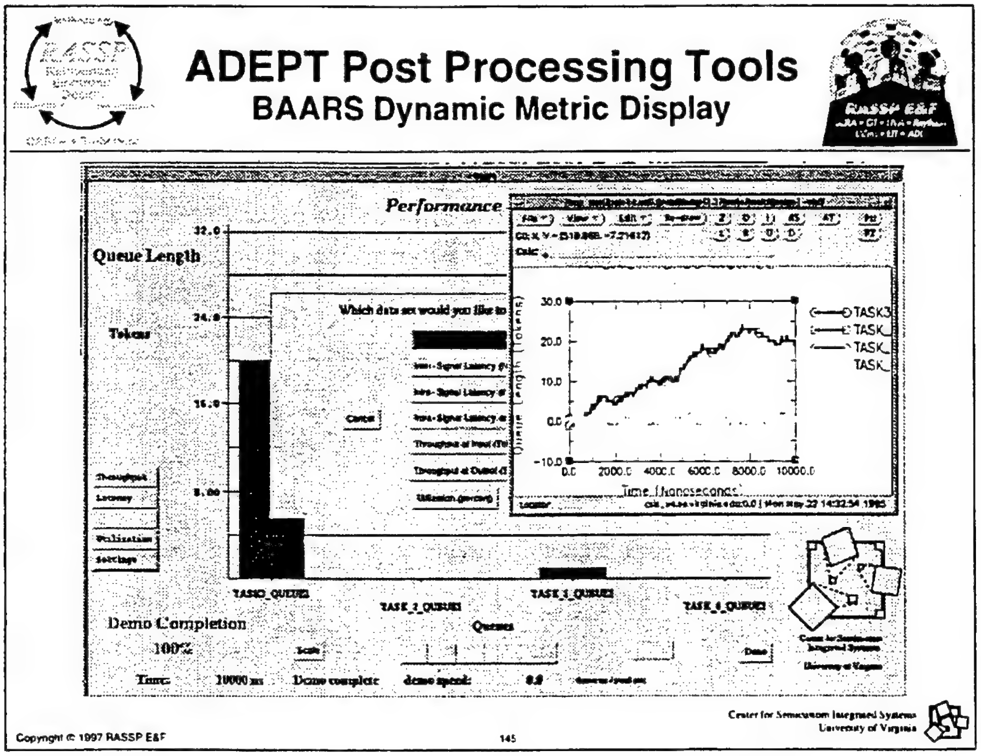
This slide shows how the actual ADEPT tools fit together with the various intermediate formats. Unfortunately, not all tools are available in all versions of ADEPT. Specifically, only the EDIF to structural VHDL path is supported on the PC platform with the OrCAD Capture tool.



## ADEPT Schematic Capture

Center for Semicustom Integrated Systems
University of Virginia

This screen shot shows the construction of an ADEPT schematic within Design Architect. Notice that all of the ADEPT utilities for constructing, simulation, and analyzing the results of an ADEPT model are available via pull-down menus.

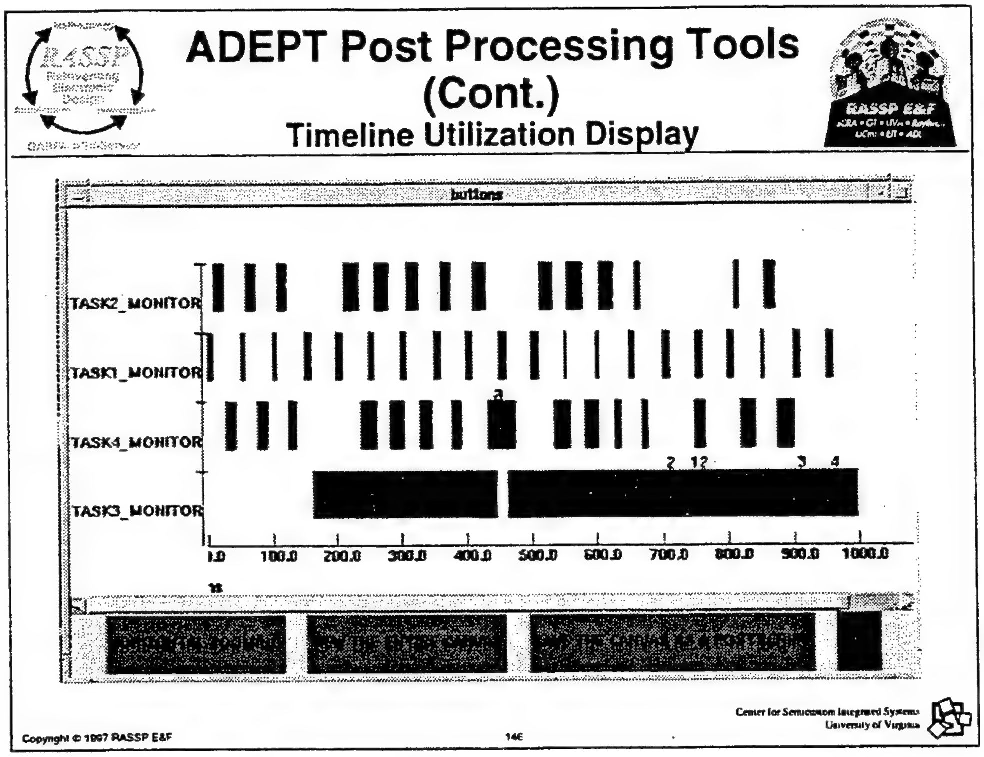


## ADEPT Post Processing Tools

### BAARS Dynamic Metric Display

Center for Semicustom Integrated Systems
University of Virginia

This is a screen shot of one of the available ADEPT post processing tools. This tool will give the user a dynamic playback of queue lengths, and module latency, utilization, and throughput over simulation time and then graph the results.



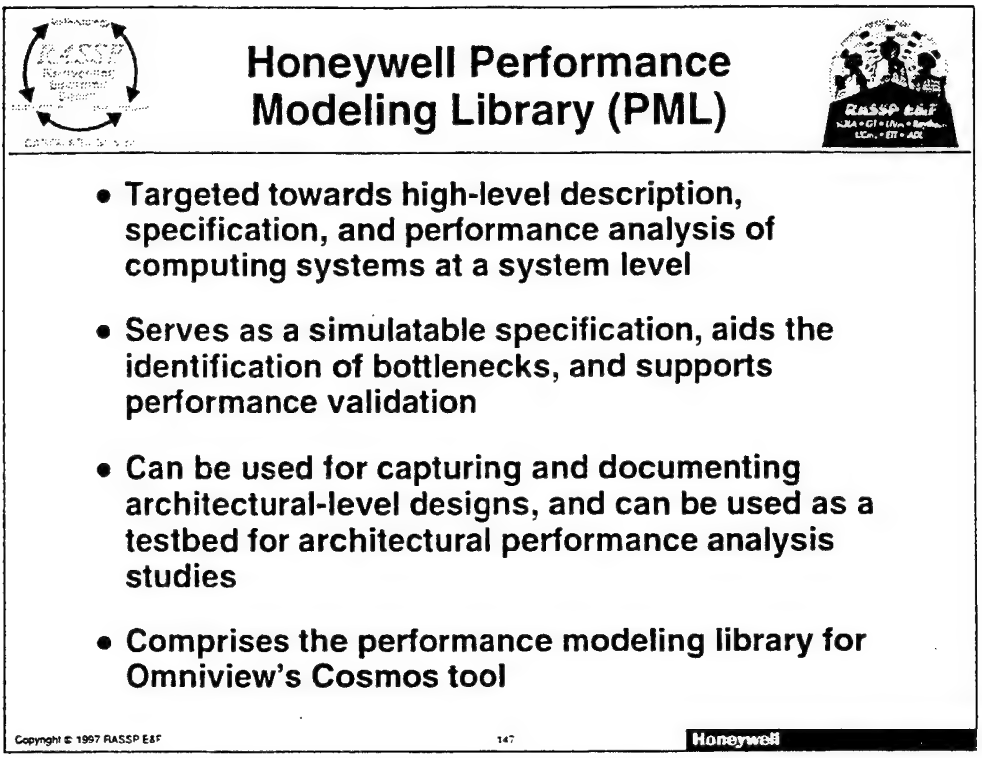
# ADEPT Post Processing Tools (Cont.)

## Timeline Utilization Display

Center for Semicustom Integrated Systems
University of Virginia

146

This is a screen shot of another of the available ADEPT post processing tools. This tool presents utilization as a standard timeline display.

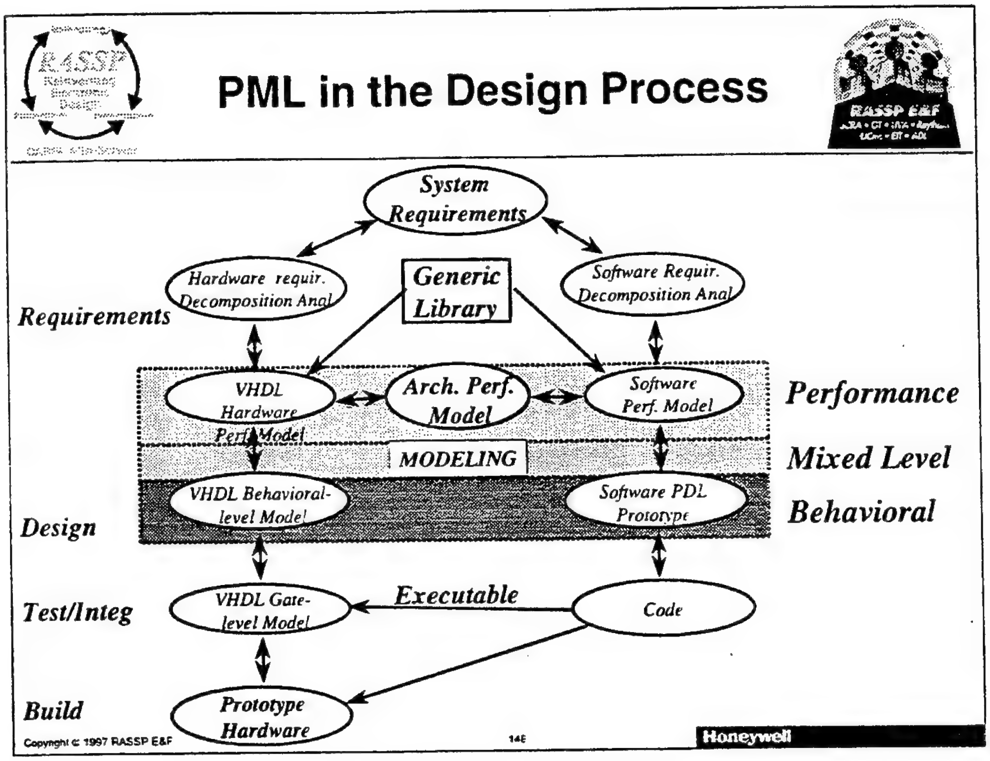


## Honeywell Performance Modeling Library (PML)

- Targeted towards high-level description, specification, and performance analysis of computing systems at a system level
- Serves as a simulatable specification, aids the identification of bottlenecks, and supports performance validation
- Can be used for capturing and documenting architectural-level designs, and can be used as a testbed for architectural performance analysis studies
- Comprises the performance modeling library for Omniview's Cosmos tool

Honeywell

Now the Performance Modeling Library (PML) developed by Honeywell Technology Center in Minneapolis MN will be discussed. PML is a VHDL-based performance modeling library of elements targeted towards modeling a system at the processor-memory-switch level. It allows the modeling and simulation of the system's hardware and software. PML is the basis of the Omniview Cosmos performance modeling tool.



## PML in the Design Process

System Requirements

Hardware requir. Decomposition Anal

Generic Library

Software Requir. Decomposition Anal

Requirements

VHDL Hardware Perf. Model

Arch. Perf. Model

Software Perf. Model

Performance

MODELING

Mixed Level

VHDL Behavioral-level Model

Software PDL Prototype

Behavioral

Design

Test/Integ

VHDL Gate-level Model

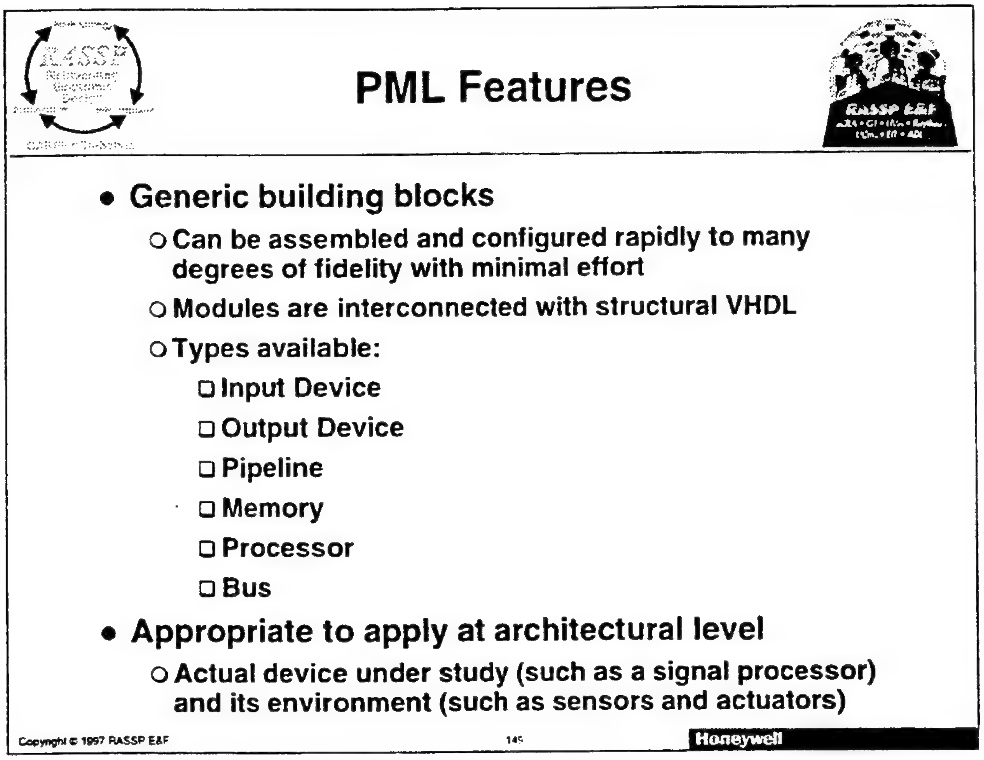
Executable

Code

Build

Prototype Hardware

This figure illustrates where the PML (and Cosmos) are intended to be used in the design process. Note that a capability for mixed level modeling (explained in the next section) is built into PML/Cosmos.

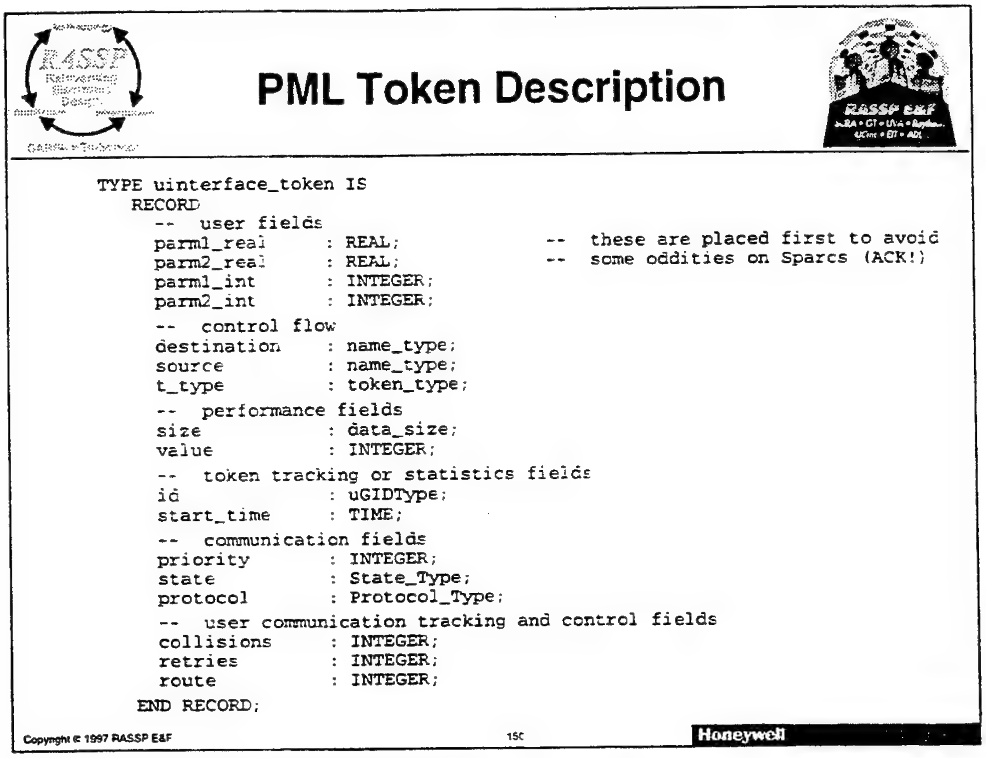


## PML Features

- **Generic building blocks**
  - ○ Can be assembled and configured rapidly to many degrees of fidelity with minimal effort
  - ○ Modules are interconnected with structural VHDL
  - ○ Types available:
    - ☐ Input Device
    - ☐ Output Device
    - ☐ Pipeline
    - ☐ Memory
    - ☐ Processor
    - ☐ Bus
- **Appropriate to apply at architectural level**
  - ○ Actual device under study (such as a signal processor) and its environment (such as sensors and actuators)

Honeywell

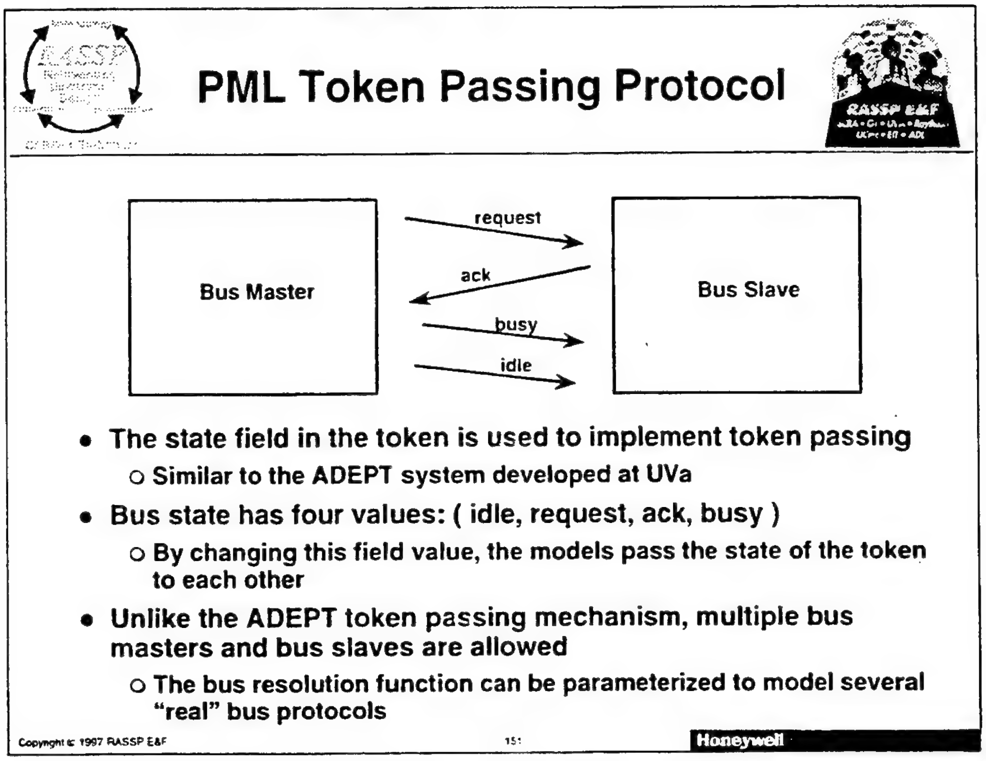
The overall approach in PML was to develop a small library of generic building blocks with many generic inputs that allowed them to be parameterized to model many different devices. The library actually contains only 5 modules and several different bus resolution functions to model communications protocols. These devices are targeted at modeling the architectural (PMS) level.



## PML Token Description

```
TYPE uinterface_token IS
   RECORD
     --  user fields
     parm1_real      : REAL;            --  these are placed first to avoid
     parm2_real      : REAL;            --  some oddities on Sparcs (ACK!)
     parm1_int       : INTEGER;
     parm2_int       : INTEGER;
     --  control flow
     destination     : name_type;
     source          : name_type;
     t_type          : token_type;
     --  performance fields
     size            : data_size;
     value           : INTEGER;
     --  token tracking or statistics fields
     id              : uGIDType;
     start_time      : TIME;
     --  communication fields
     priority        : INTEGER;
     state           : State_Type;
     protocol        : Protocol_Type;
     --  user communication tracking and control fields
     collisions      : INTEGER;
     retries         : INTEGER;
     route           : INTEGER;
   END RECORD;
```

Here is a description of the generic token defined by Honeywell Technology Center for interoperability of performance models [HTC97]. The actual token used inside of PML is proprietary and slightly different than this, but this example gives the overall structure and how it is different from the ADEPT token.

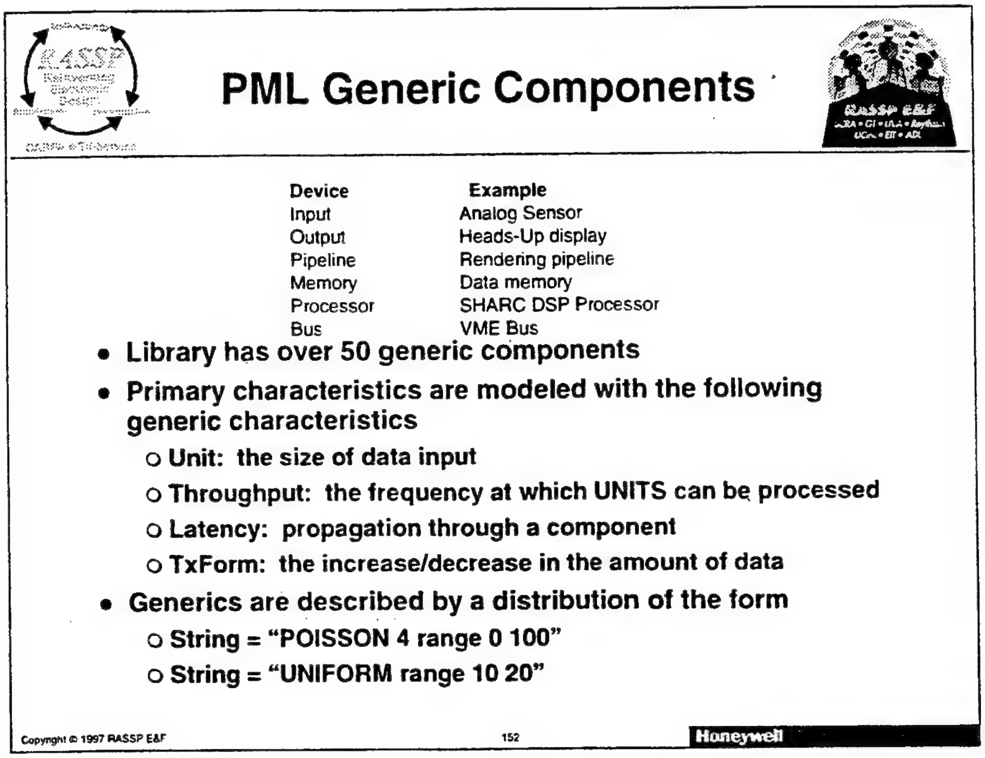


## PML Token Passing Protocol

Bus Master
request
ack
busy
idle
Bus Slave

- **The state field in the token is used to implement token passing**
  - **Similar to the ADEPT system developed at UVa**
- **Bus state has four values: ( idle, request, ack, busy )**
  - **By changing this field value, the models pass the state of the token to each other**
- **Unlike the ADEPT token passing mechanism, multiple bus masters and bus slaves are allowed**
  - **The bus resolution function can be parameterized to model several "real" bus protocols**

The VHDL bus resolution function (BRF) used in PML uses four states to pass tokens on busses that have multiple drives and sources. For simple point-to-point connections, only three states are used for simulation efficiency. The BRF can be parameterized (or modified) to model several "real" bus protocols - thus the VHDL BRF is actually part of the model.



## PML Generic Components

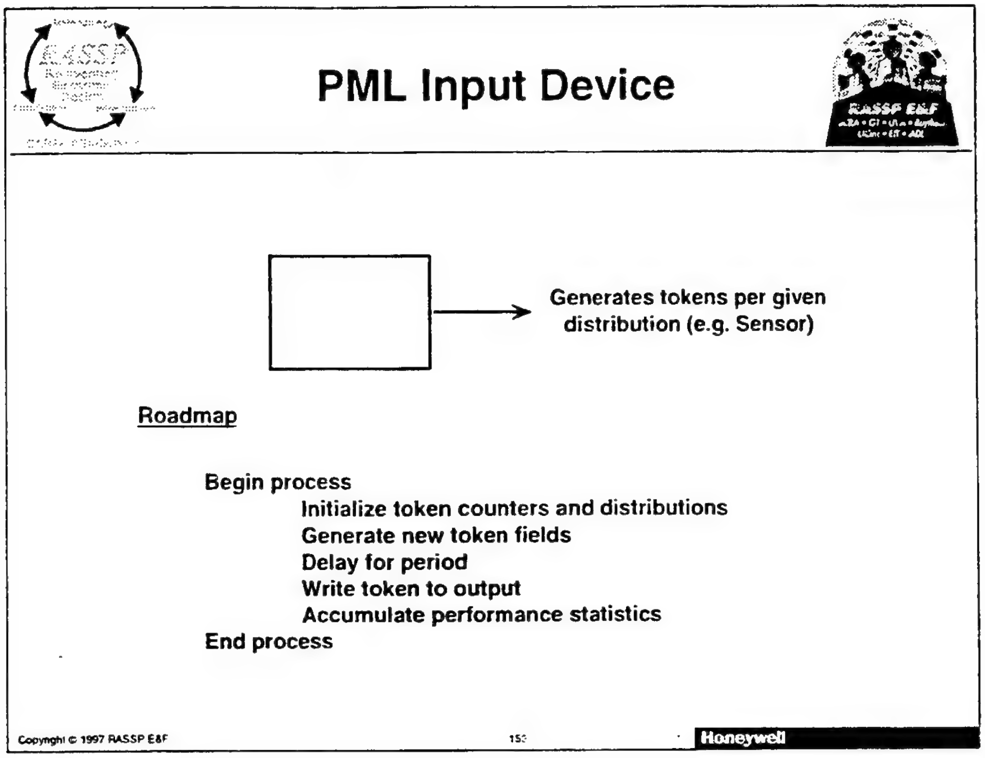
| Device | Example |
|---|---|
| Input | Analog Sensor |
| Output | Heads-Up display |
| Pipeline | Rendering pipeline |
| Memory | Data memory |
| Processor | SHARC DSP Processor |
| Bus | VME Bus |

- **Library has over 50 generic components**
- **Primary characteristics are modeled with the following generic characteristics**
  - **Unit: the size of data input**
  - **Throughput: the frequency at which UNITS can be processed**
  - **Latency: propagation through a component**
  - **TxForm: the increase/decrease in the amount of data**
- **Generics are described by a distribution of the form**
  - **String = "POISSON 4 range 0 100"**
  - **String = "UNIFORM range 10 20"**

Honeywell

As stated previously, the PML library consists of 5 major modules, but there are many examples of modules parameterized to model specific devices in the library.

PML contains a sophisticated string processing language for specification of complex generic parameters to the models.



## PML Input Device

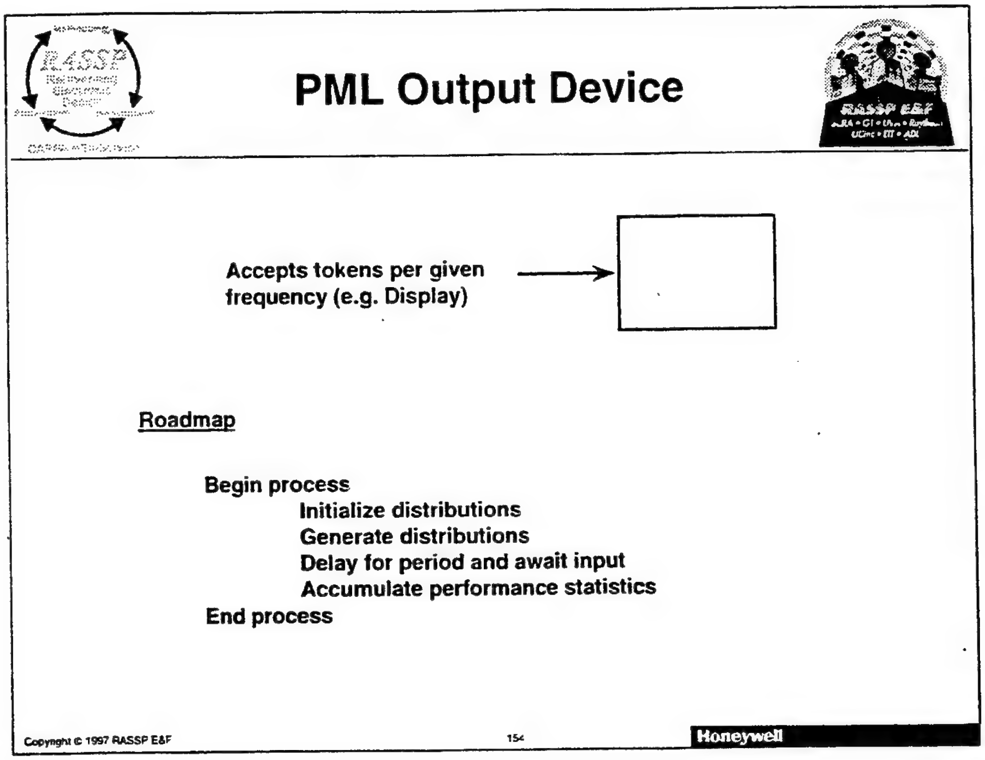
Generates tokens per given distribution (e.g. Sensor)

<u>Roadmap</u>

```
Begin process
        Initialize token counters and distributions
        Generate new token fields
        Delay for period
        Write token to output
        Accumulate performance statistics
End process
```

Honeywell

A PML input device is like a Source module in ADEPT, it creates tokens at a specified rate. Note that all modules in PML participate in the generation of performance statistics like latency and utilization.



## PML Output Device

Accepts tokens per given frequency (e.g. Display) →

**Roadmap**

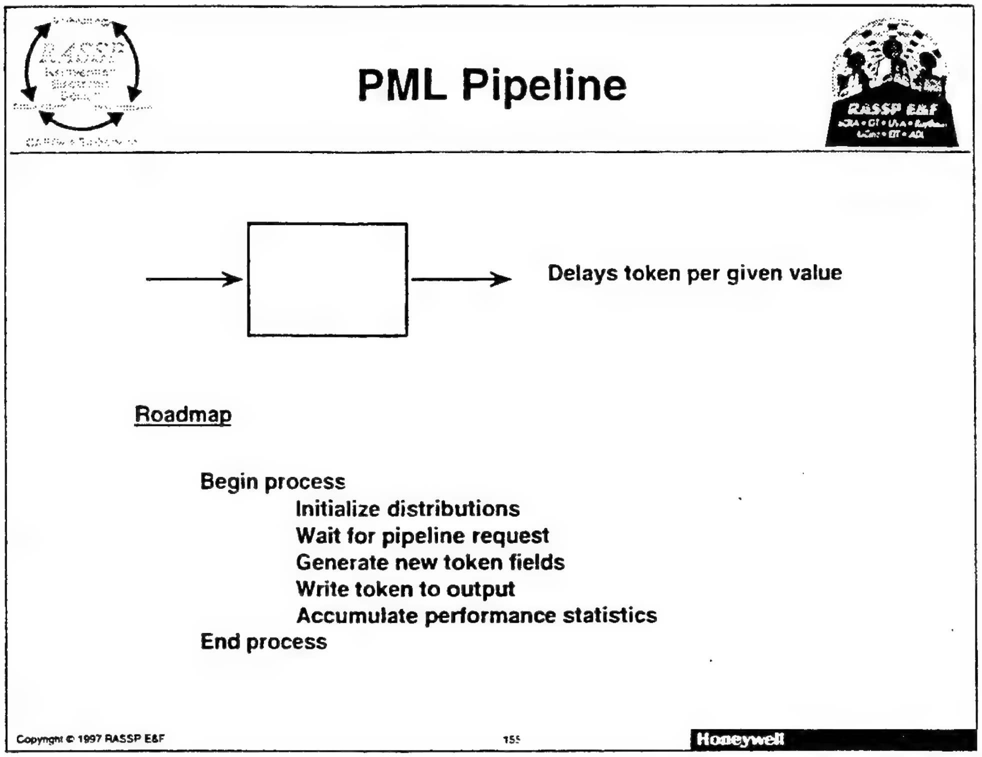
```
Begin process
        Initialize distributions
        Generate distributions
        Delay for period and await input
        Accumulate performance statistics
End process
```

Copyright © 1997 RASSP E&F

154

Honeywell

An output device is like a Sink module in ADEPT. It consumes tokens.



## PML Pipeline

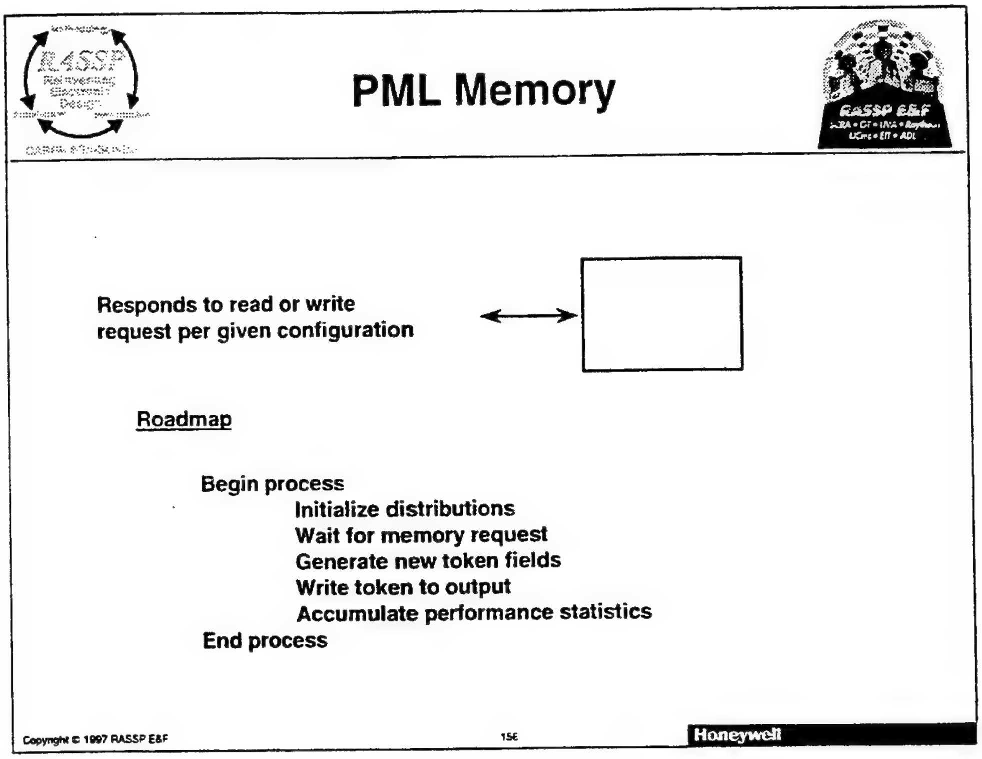
Delays token per given value

**Roadmap**

```
Begin process
        Initialize distributions
        Wait for pipeline request
        Generate new token fields
        Write token to output
        Accumulate performance statistics
End process
```

Honeywell

The pipeline component delays tokens. It can also, by changing token fields, route tokens.



## PML Memory

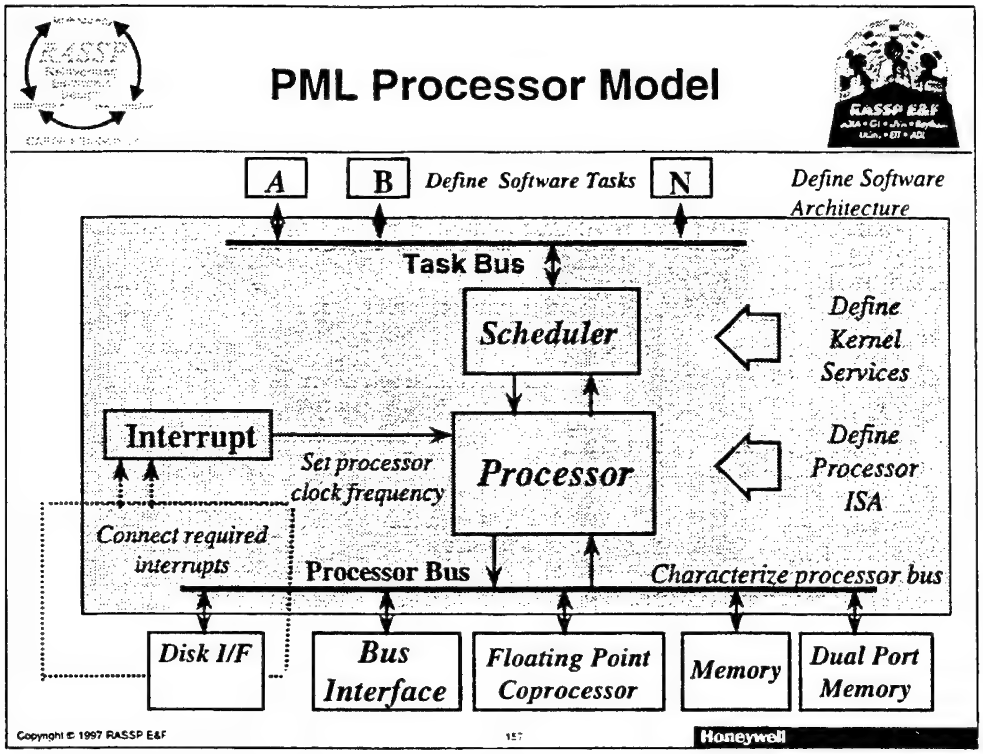
Responds to read or write request per given configuration

Roadmap

```
Begin process
        Initialize distributions
        Wait for memory request
        Generate new token fields
        Write token to output
        Accumulate performance statistics
End process
```

Honeywell

The memory component consumes memory request tokens and after a specified delay, generates memory access tokens.



## PML Processor Model

Copyright © 1997 RASSP E&F

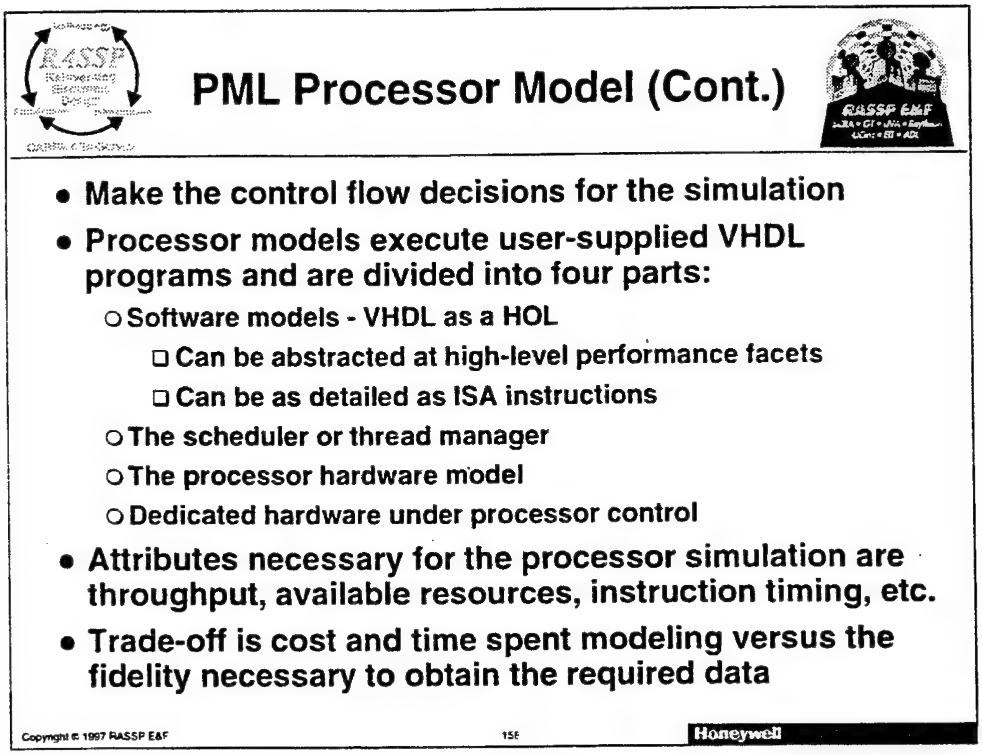
157

Honeywell

The processor model is the heart of the PML. It is capable of running a representation of the software that the real system will execute. That software representation, while written in VHDL can be at a level of abstraction that ranges from the task level down to the detailed functional level.

The PML processor is basically a request-resource model. The software representation executes and a specified point, requests resources (e.g. memory access, 1000 floating point multiplies, 100 integer adds, etc.) from the processor. The processor schedules these operations on the hardware resource when it is available and delays the software execution until they are completed. The software continues from that point until more hardware resources are needed.

The processor is parameterized by specifying its Instruction Set Architecture (ISA) and what and how many resources are consumed by each instruction in the ISA. Sophisticated operating system constructs such as interrupts and multitasking can be modeled as well.

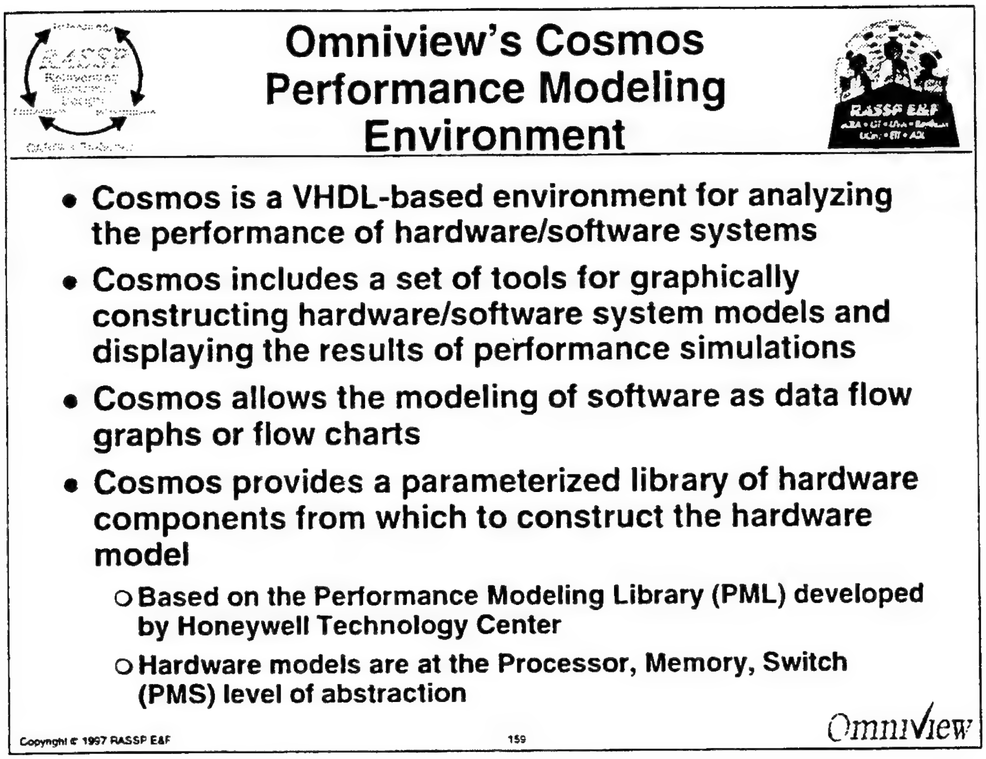


## PML Processor Model (Cont.)

- Make the control flow decisions for the simulation
- Processor models execute user-supplied VHDL programs and are divided into four parts:
  - ○ Software models - VHDL as a HOL
    - ☐ Can be abstracted at high-level performance facets
    - ☐ Can be as detailed as ISA instructions
  - ○ The scheduler or thread manager
  - ○ The processor hardware model
  - ○ Dedicated hardware under processor control
- Attributes necessary for the processor simulation are throughput, available resources, instruction timing, etc.
- Trade-off is cost and time spent modeling versus the fidelity necessary to obtain the required data

Honeywell

The processor model allows detailed modeling of software at various levels of abstraction executing on different types and speeds of processors. One drawback of this fidelity (and its associated complexity) is long simulation times.



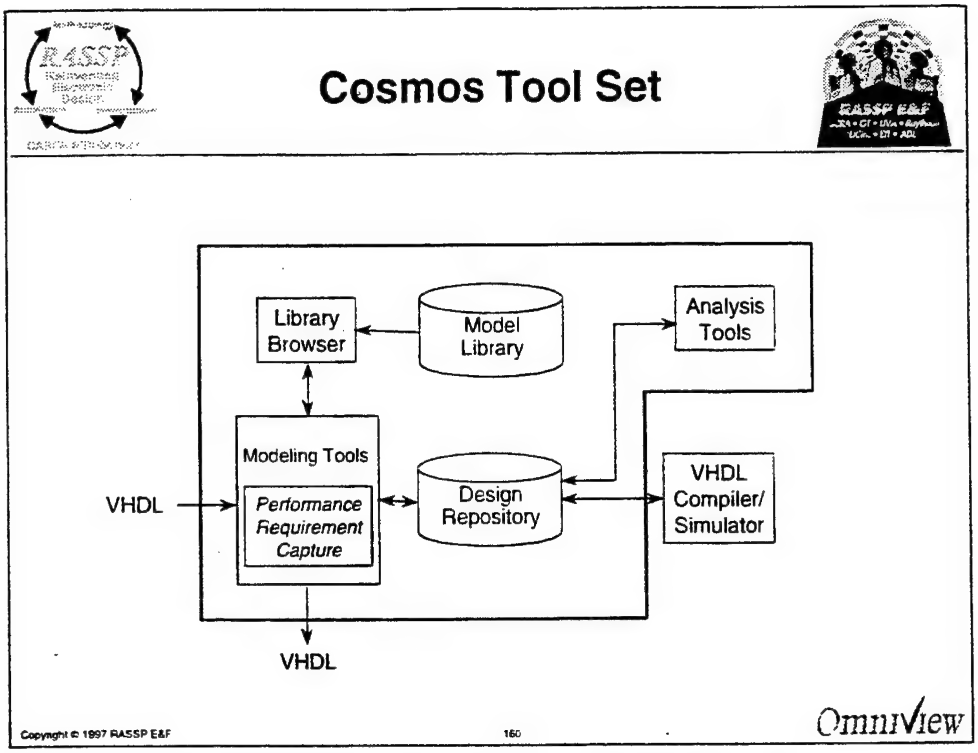
## Omniview's Cosmos Performance Modeling Environment

- Cosmos is a VHDL-based environment for analyzing the performance of hardware/software systems
- Cosmos includes a set of tools for graphically constructing hardware/software system models and displaying the results of performance simulations
- Cosmos allows the modeling of software as data flow graphs or flow charts
- Cosmos provides a parameterized library of hardware components from which to construct the hardware model
  - ○ Based on the Performance Modeling Library (PML) developed by Honeywell Technology Center
  - ○ Hardware models are at the Processor, Memory, Switch (PMS) level of abstraction

OmniView

This section describes Omniview's Cosmos tool. Cosmos is very ADEPT like in that it includes tools for constructing, simulating, and analyzing performance models in VHDL. It uses the Performance Modeling Library (PML) developed by Honeywell Technology Center as its module library. The development of Cosmos was funded as part of the RASSP program.

Note that unlike ADEPT, Cosmos (like PML) is targeted at one specific level of performance modeling (the processor, memory, switch (PMS) level) and does not have a mathematical foundation or support dependability analysis.



## Cosmos Tool Set

Library Browser

Model Library

Analysis Tools

Modeling Tools

*Performance Requirement Capture*

Design Repository

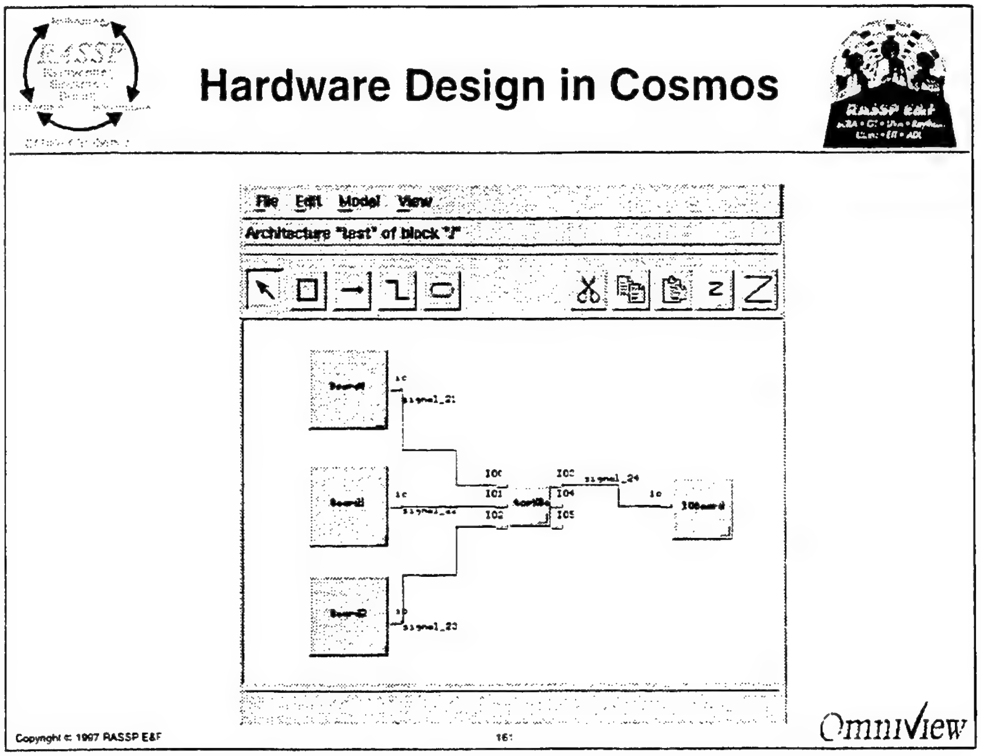
VHDL Compiler/ Simulator

VHDL

VHDL

OmniView

This is the Cosmos tool set. Like ADEPT, a commercial, third party, VHDL simulator is used as the simulation engine and must be obtained separately.



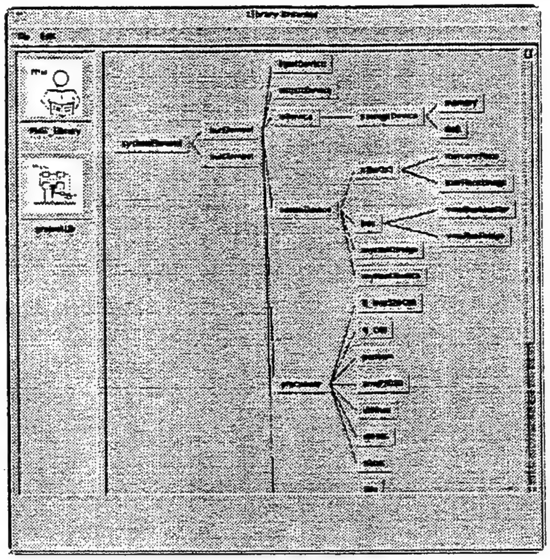
## Hardware Design in Cosmos

161

OmniView

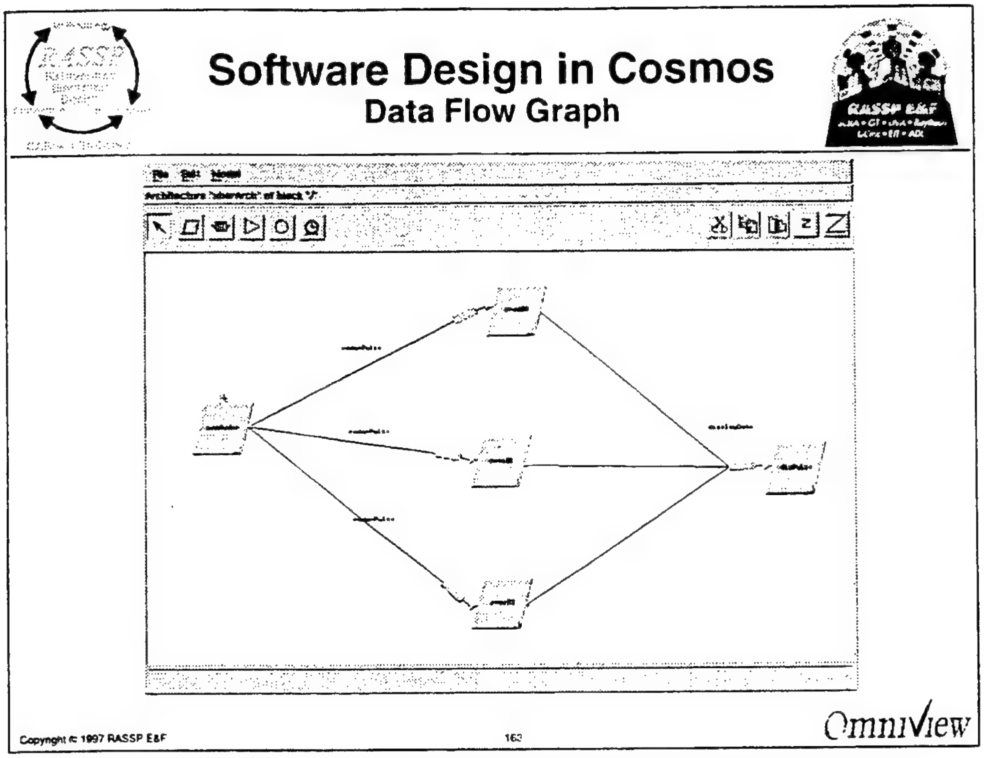
This is an illustration of the construction of the hardware model in Cosmos. The hardware model consists of processor models and communications switch models from the PML library (as will be presented). The modules used in the model can be parameterized, via the GUI, to model different types of processors and networks.

## Cosmos Library Browser



OmniView

This is the COSMOS library browser. It is used to select standard hardware components out of the library for instantiation into a performance model. COSMOS comes with the complete PML library of generic elements and several specific components (like a Mercury RaceWay crossbar switch) built out of those generic components.



## Software Design in Cosmos

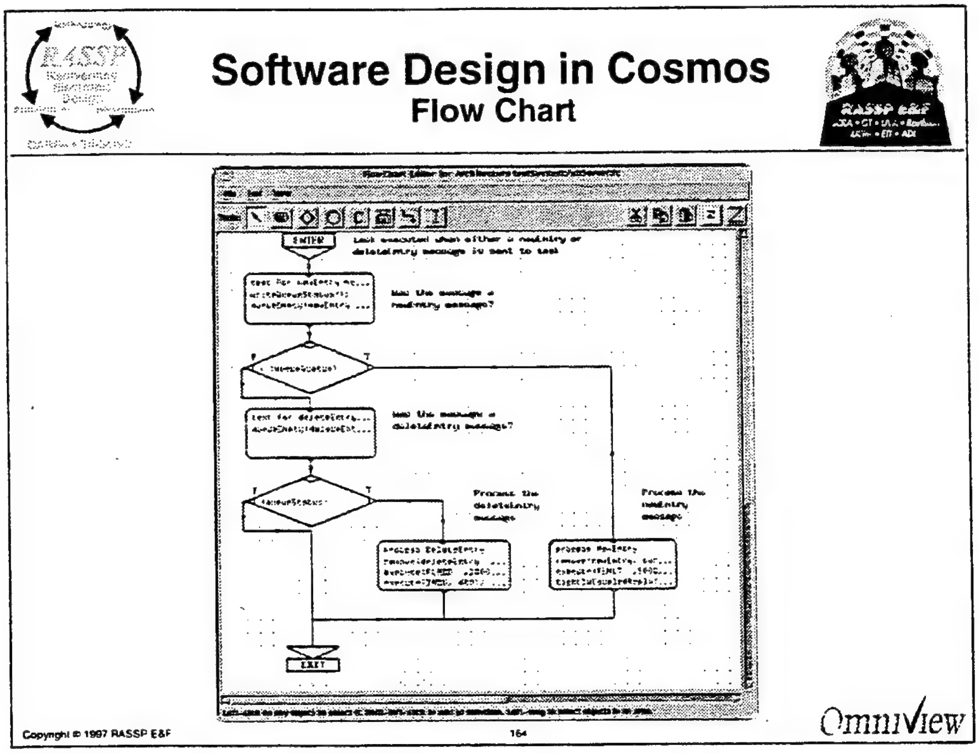
### Data Flow Graph

Copyright © 1997 RASSP E&F

163

OmniView

This is an illustration of the description of the software application in Cosmos. Here, the software is described as a dataflow graph as is common in embedded DSP applications.

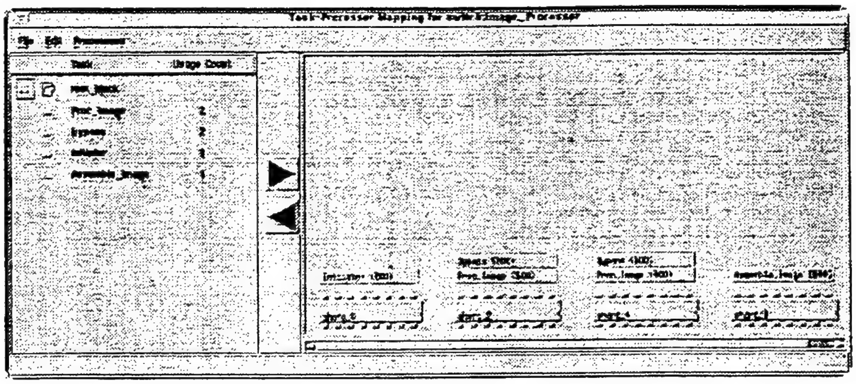


## Software Design in Cosmos
### Flow Chart

Software can also be described as a control flow graph in COSMOS as shown here.

In addition to the two methods shown in this slide and the previous one, software in COSMOS can be coded directly in VHDL by the user (with appropriate calls to the hardware resource models), and included in the COSMOS model.

# Software to Hardware Mapping in Cosmos

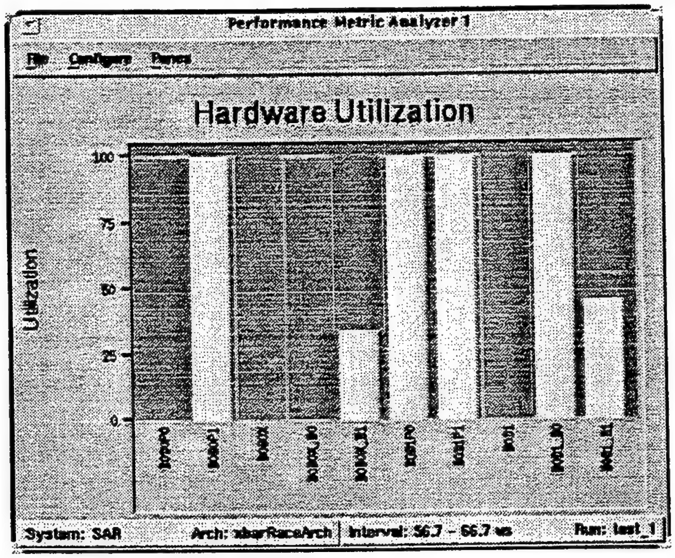


OmniView

Once the hardware and software models are completed, the next step is to map the software tasks onto specific hardware processors for execution. This is done with the software mapping tool as shown here.

# Analysis of Results in Cosmos

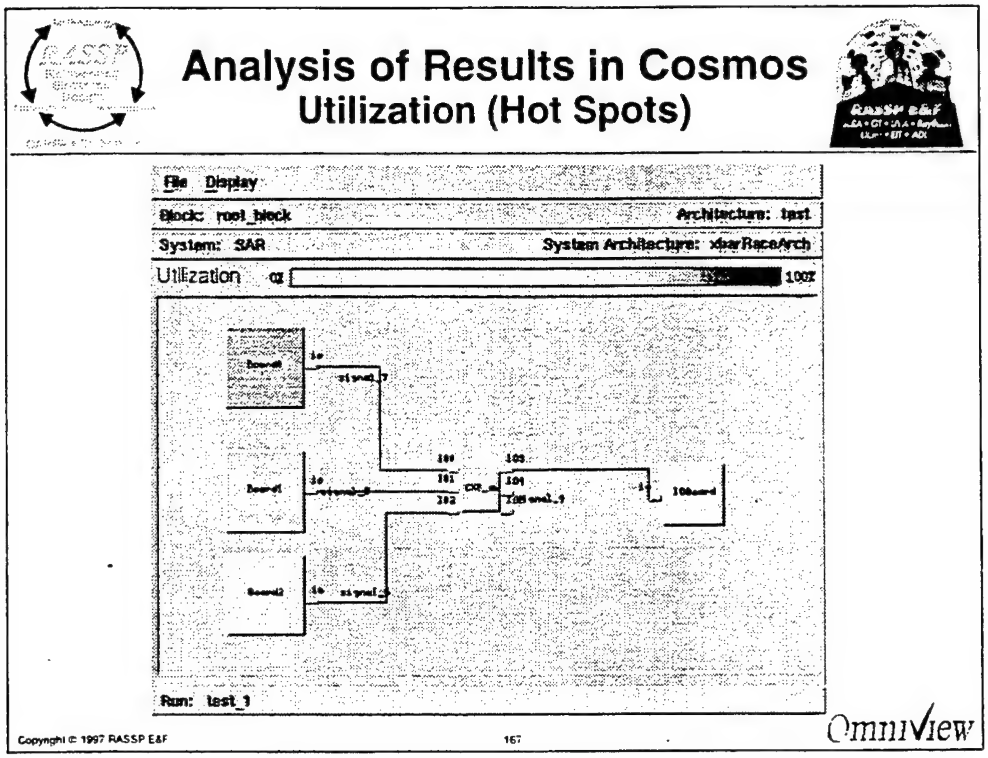
## Utilization



System: SAR Arch: xbarRaceArch Interval: 56.7 - 66.7 us Run: test_1

OmniView

Like ADEPT, COSMOS contains a number of tools for analyzing the data from the performance model simulation. This is the COSMOS utilization tool display. It displays specific processor utilization as a moving horizontal bar graph.



## Analysis of Results in Cosmos
### Utilization (Hot Spots)

File Display

Block: root_block    Architecture: test

System: SAR    System Architecture: xbarRaceArch

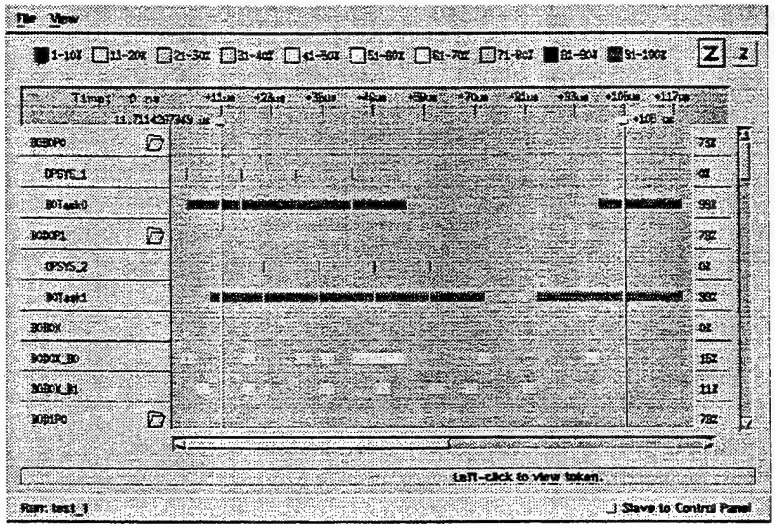
Utilization 0% 100%

Run: test_1

Copyright © 1997 RASSP E&F    167    OmniView

Here is another COSMOS post-simulation data display tool. In this case, its a "hot spot" display which show module utilization in color codes. Modules that appear towards the red side of the spectrum are highly utilized and may represent a bottleneck in the computation. If however, all modules are towards the blue side of the spectrum, the overall system may be over designed resulting in wasted resources.

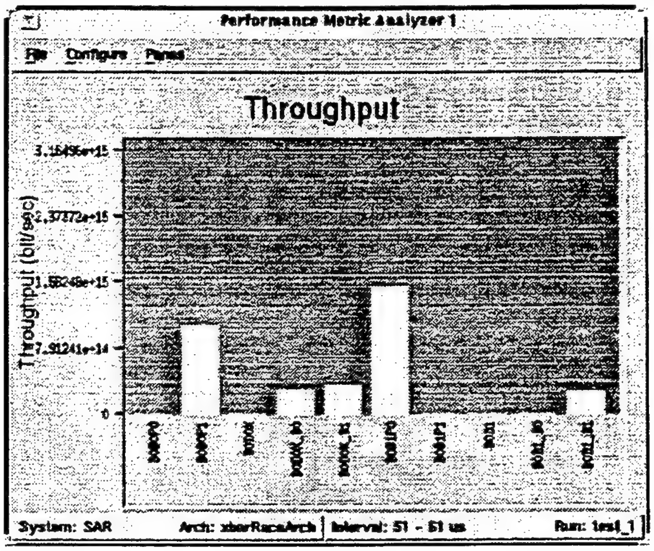
# Analysis of Results in Cosmos
## Activity Time Lines



OmniView

This is a screen shot of the activity time line display available in Cosmos. It is fairly standard.

# Analysis of Results in Cosmos
## Throughput



System: SAR

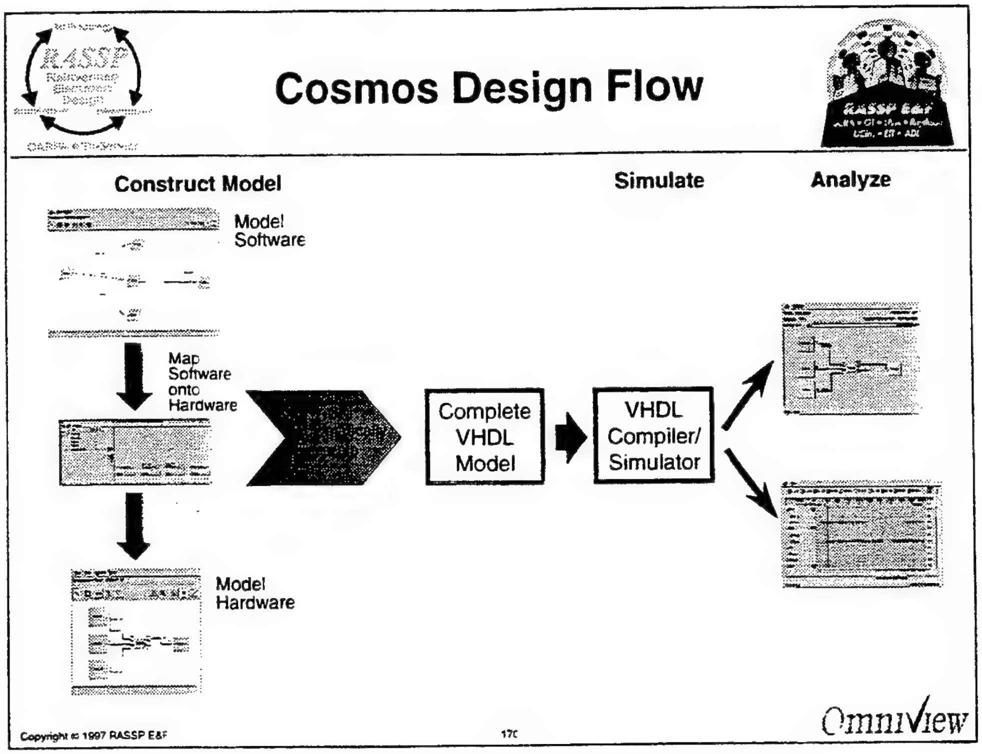
Arch: xbarRaceArch

Interval: 51 - 61 us

Run: test_1

OmniView

Here is the throughput display from COSMOS.



## Cosmos Design Flow

**Construct Model**

**Simulate**

**Analyze**

Model Software

Map Software onto Hardware

Model Hardware

Complete VHDL Model

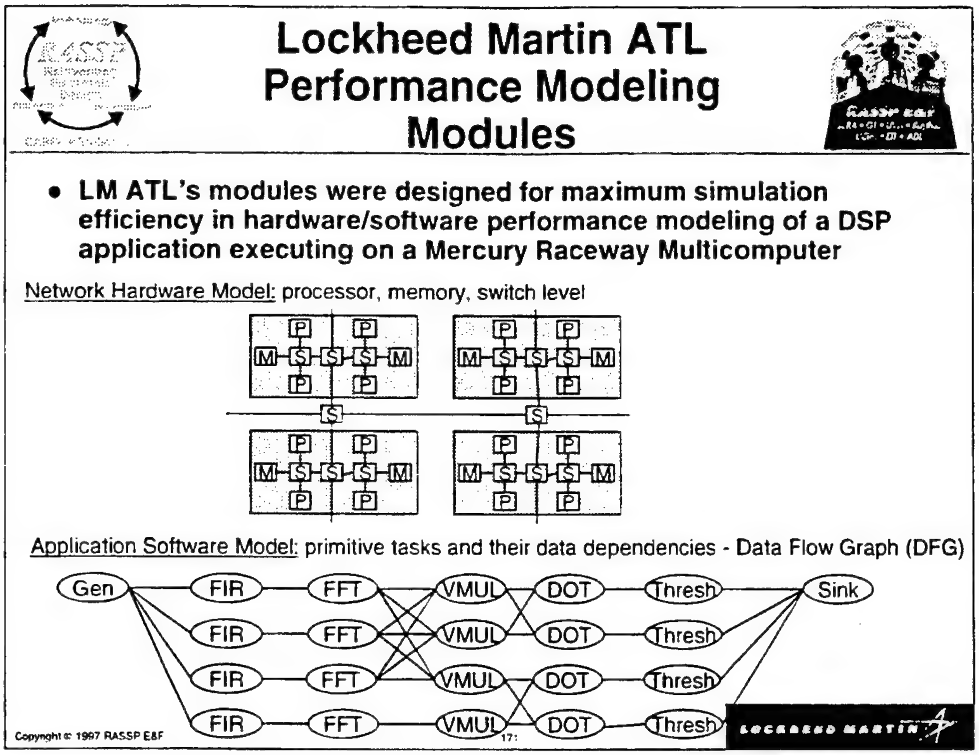
VHDL Compiler/ Simulator

Copyright © 1997 RASSP E&F

170

OmniView

This slide shows the overall design flow in COSMOS. Again, the hardware architecture is modeled using the PML library modules configured to model the chosen hardware architecture. This includes specifying the ISA of the chosen processors and their execution rates, and the network configuration and its communication rates. The software is modeled as a set of tasks that communicate in a specific way and take a certain amount of resources in terms of computation and communication. Finally, the mapping of software tasks to processors is specified. The COSMOS tools then generate a VHDL model of the complete system which is then compiled and simulated on the chosen commercial VHDL simulator. The data that results from that simulation can then be displayed graphically by the COSMOS post-simulation analysis tools.



## Lockheed Martin ATL Performance Modeling Modules

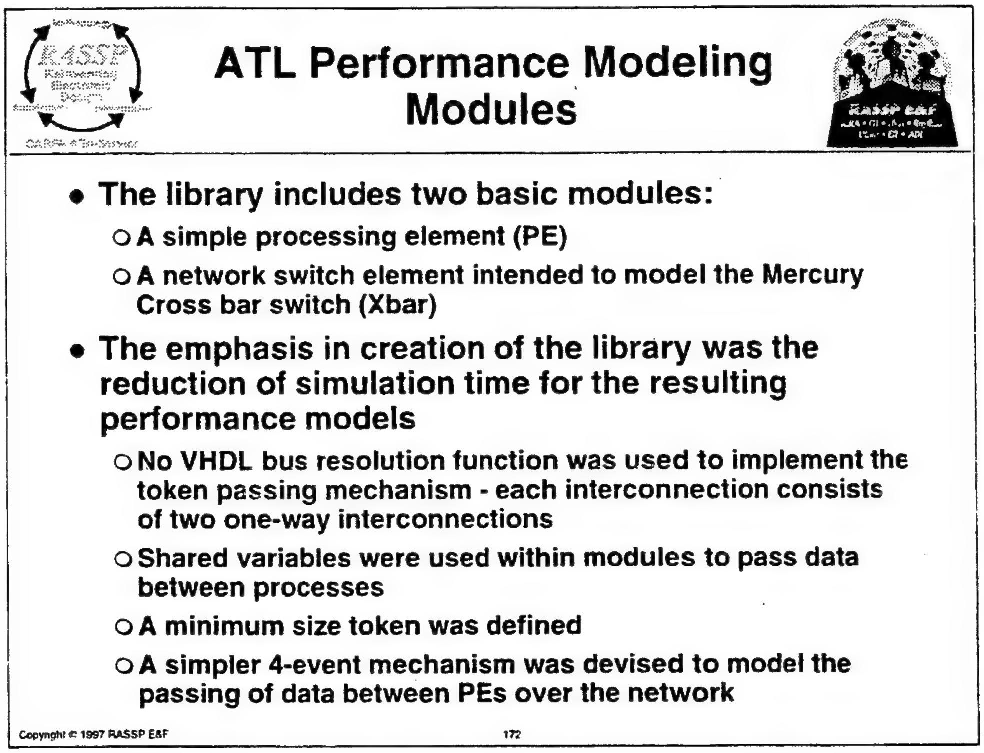
- **LM ATL's modules were designed for maximum simulation efficiency in hardware/software performance modeling of a DSP application executing on a Mercury Raceway Multicomputer**

Network Hardware Model: processor, memory, switch level

Application Software Model: primitive tasks and their data dependencies - Data Flow Graph (DFG)

As part of the RASSP program, ATL was tasked to use performance modeling in the design of several benchmark embedded DSP systems. Their efforts to use PML and ADEPT at an early point in the program were hindered by the long simulation times of both ADEPT and PML models and by the unavailability, at that time, of the COSMOS tool and a suitable PMS level modeling library in ADEPT. In response, they developed a very lightweight PMS level modeling environment for Mercury Raceway systems with an emphasis on reduced simulation times.

Note that both ADEPT and PML have since addressed the simulation time problem with good results.

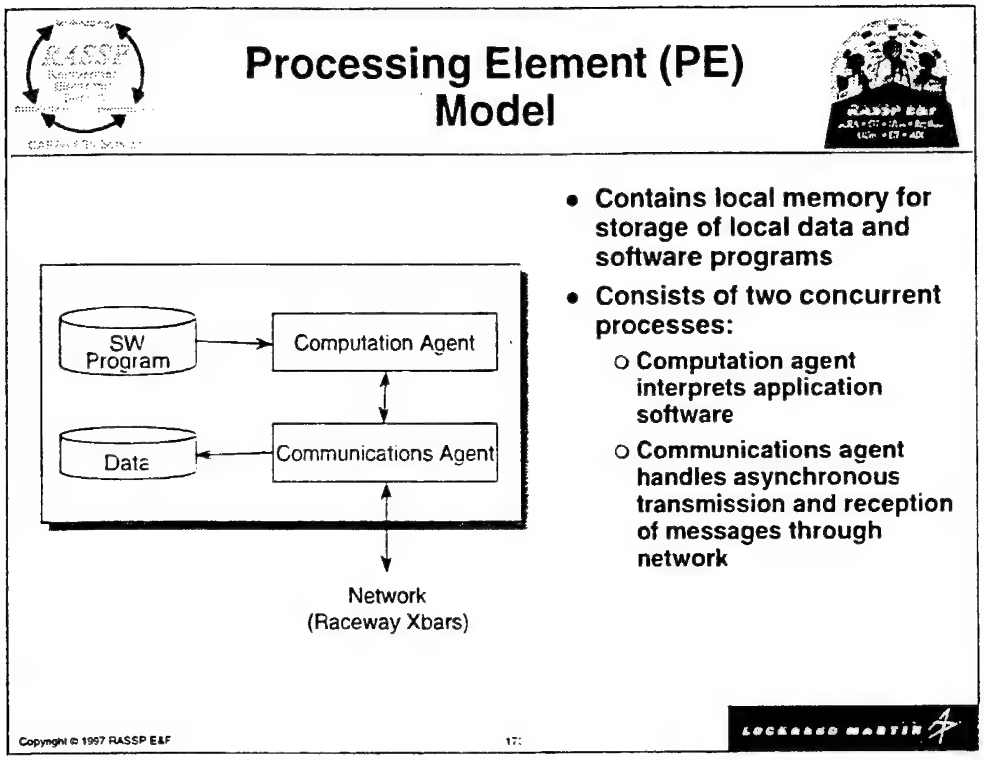


## ATL Performance Modeling Modules

- **The library includes two basic modules:**
  - ○ A simple processing element (PE)
  - ○ A network switch element intended to model the Mercury Cross bar switch (Xbar)
- **The emphasis in creation of the library was the reduction of simulation time for the resulting performance models**
  - ○ No VHDL bus resolution function was used to implement the token passing mechanism - each interconnection consists of two one-way interconnections
  - ○ Shared variables were used within modules to pass data between processes
  - ○ A minimum size token was defined
  - ○ A simpler 4-event mechanism was devised to model the passing of data between PEs over the network

The ATL library consists of two components, a processor model (which includes a network interface), and a switch model. The switch is intended to model the Mercury Raceway crossbar switch.

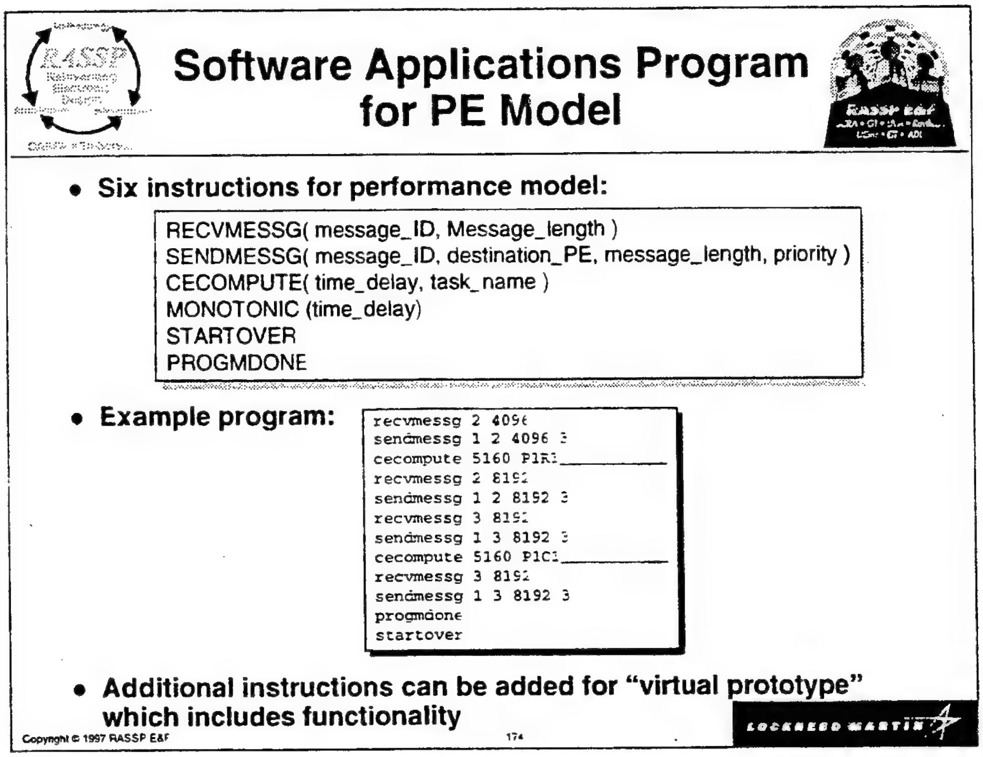
Much emphasis was placed on reducing simulation times and the results were very good in that regard - ATL VHDL performance models of the Raceway system simulate in an equivalent time to models written in C. However, the disadvantage of this more ad hoc approach over ADEPT or PML is the limited library of components available (which had to be written specifically for this network model) and a less general applicability.


SW Program
Computation Agent
Data
Communications Agent
Network
(Raceway Xbars)

## Processing Element (PE) Model

- **Contains local memory for storage of local data and software programs**
- **Consists of two concurrent processes:**
  - **o Computation agent interprets application software**
  - **o Communications agent handles asynchronous transmission and reception of messages through network**

The ATL processing element (PE) consists of two parts; the computation agent that reads CPU instruction from a file and executes them, and a communications agent that interfaces to the network model and handles message sends and receives.



## Software Applications Program for PE Model

- **Six instructions for performance model:**

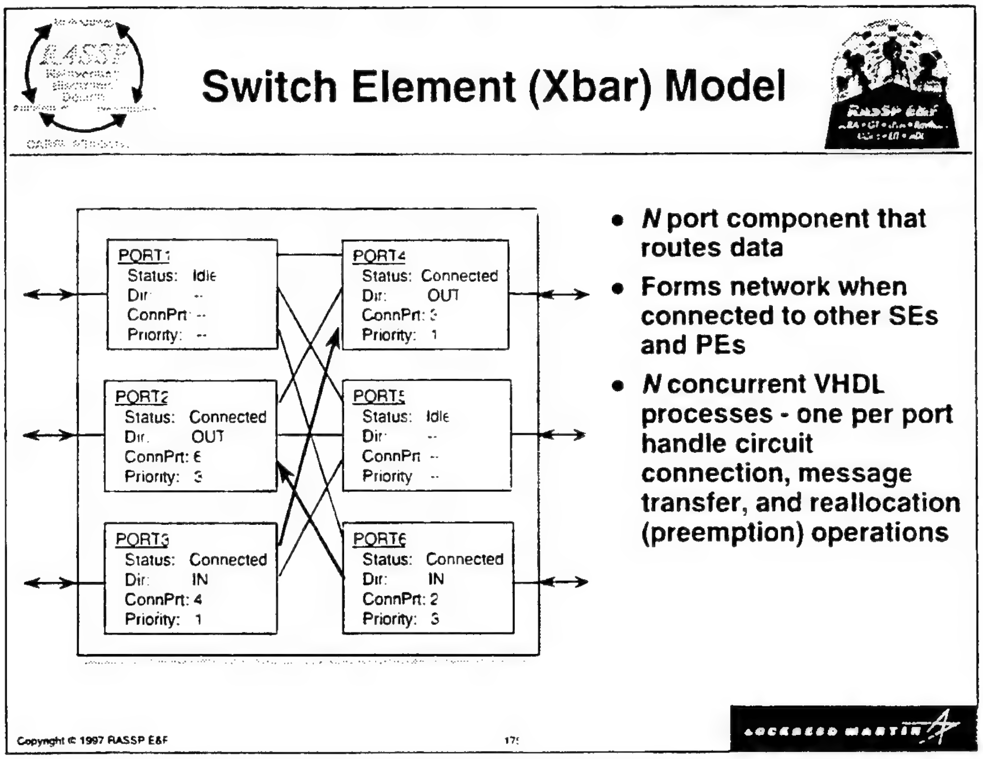
```
RECVMESSG( message_ID, Message_length )
SENDMESSG( message_ID, destination_PE, message_length, priority )
CECOMPUTE( time_delay, task_name )
MONOTONIC (time_delay)
STARTOVER
PROGMDONE
```

- **Example program:**

```
recvmessg 2 4096
sendmessg 1 2 4096 3
cecompute 5160 P1R1
recvmessg 2 8192
sendmessg 1 2 8192 3
recvmessg 3 8192
sendmessg 1 3 8192 3
cecompute 5160 P1C1
recvmessg 3 8192
sendmessg 1 3 8192 3
progmdone
startover
```

- **Additional instructions can be added for "virtual prototype" which includes functionality**

The ATL CPU model has 6 instructions that fall into three basic modes, compute, send and receive. Additional instructions that perform actual data translations (complex multiply, matrix operations, etc.) can be added in the first "virtual prototype" stage when some functionality is added to the model.



## Switch Element (Xbar) Model

PORT1
Status: Idle
Dir: --
ConnPrt: --
Priority: --

PORT2
Status: Connected
Dir: OUT
ConnPrt: 6
Priority: 3

PORT3
Status: Connected
Dir: IN
ConnPrt: 4
Priority: 1

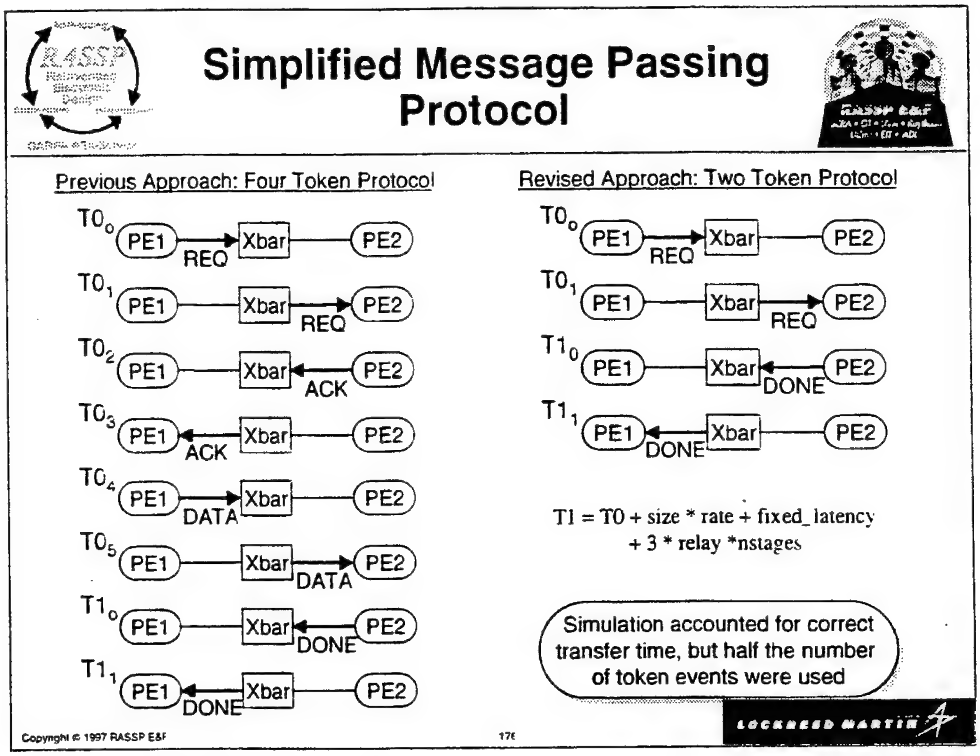
PORT4
Status: Connected
Dir: OUT
ConnPrt: 3
Priority: 1

PORT5
Status: Idle
Dir: --
ConnPrt: --
Priority: --

PORT6
Status: Connected
Dir: IN
ConnPrt: 2
Priority: 3

- ***N*** **port component that routes data**
- **Forms network when connected to other SEs and PEs**
- ***N*** **concurrent VHDL processes - one per port handle circuit connection, message transfer, and reallocation (preemption) operations**

The switch element in the ATL library models a 6 port Mercury Raceway crossbar switch. This crossbar is circuit switched and can handle up to three simultaneous connections. It is modeled in VHDL using 6 concurrent VHDL processes, one to handle each port on the crossbar. The crossbar functions of circuit setup, teardown and preemption are handled.



## Simplified Message Passing Protocol

Previous Approach: Four Token Protocol

$T0_0$ PE1 → Xbar — PE2 REQ
$T0_1$ PE1 — Xbar → PE2 REQ
$T0_2$ PE1 — Xbar ← PE2 ACK
$T0_3$ PE1 ← Xbar — PE2 ACK
$T0_4$ PE1 → Xbar — PE2 DATA
$T0_5$ PE1 — Xbar → PE2 DATA
$T1_0$ PE1 — Xbar ← PE2 DONE
$T1_1$ PE1 ← Xbar — PE2 DONE

Revised Approach: Two Token Protocol

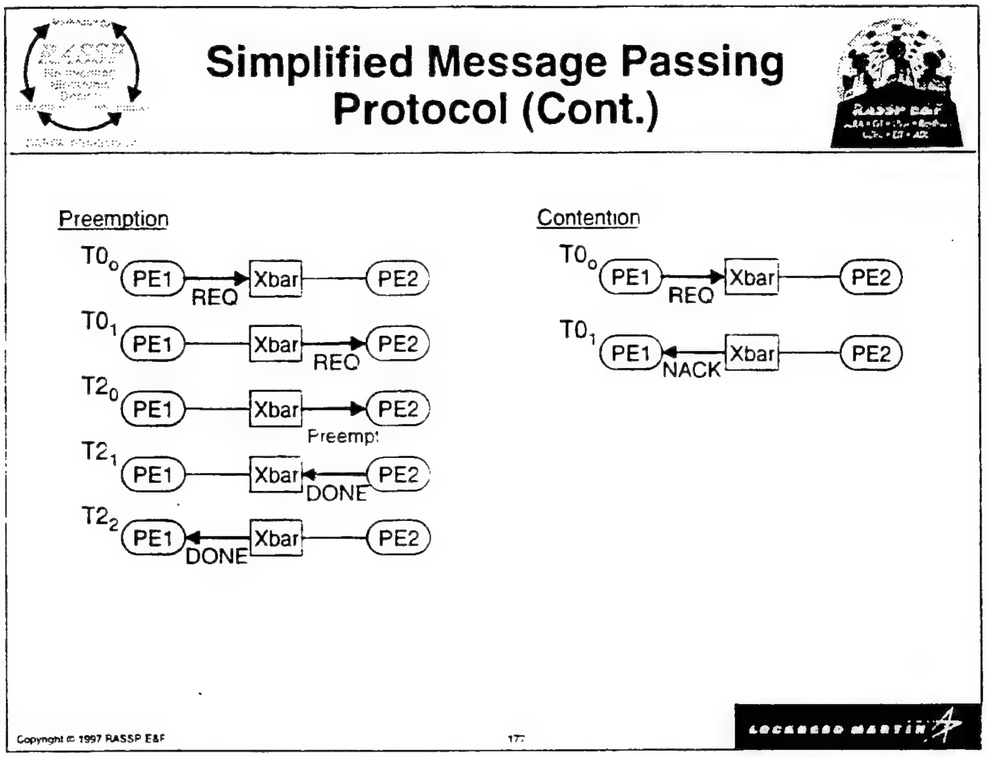
$T0_0$ PE1 → Xbar — PE2 REQ
$T0_1$ PE1 — Xbar → PE2 REQ
$T1_0$ PE1 — Xbar ← PE2 DONE
$T1_1$ PE1 ← Xbar — PE2 DONE

$$T1 = T0 + size * rate + fixed\_latency + 3 * relay * nstages$$

Simulation accounted for correct transfer time, but half the number of token events were used

This is an illustration of how the normal message passing protocol, as modeled in a performance modeling environment, was simplified to reduce the number of tokens needed. Note that this token passing mechanism is a modeling artifact, it is not how the Raceway actually passes data, so changing it does not affect the model fidelity as long as care is taken to keep the timing the same.

Also note that the ATL module do not use bus resolution functions to pass tokens - they use two unidirectional signals - further decreasing the execution time of the simulation.



## Simplified Message Passing Protocol (Cont.)

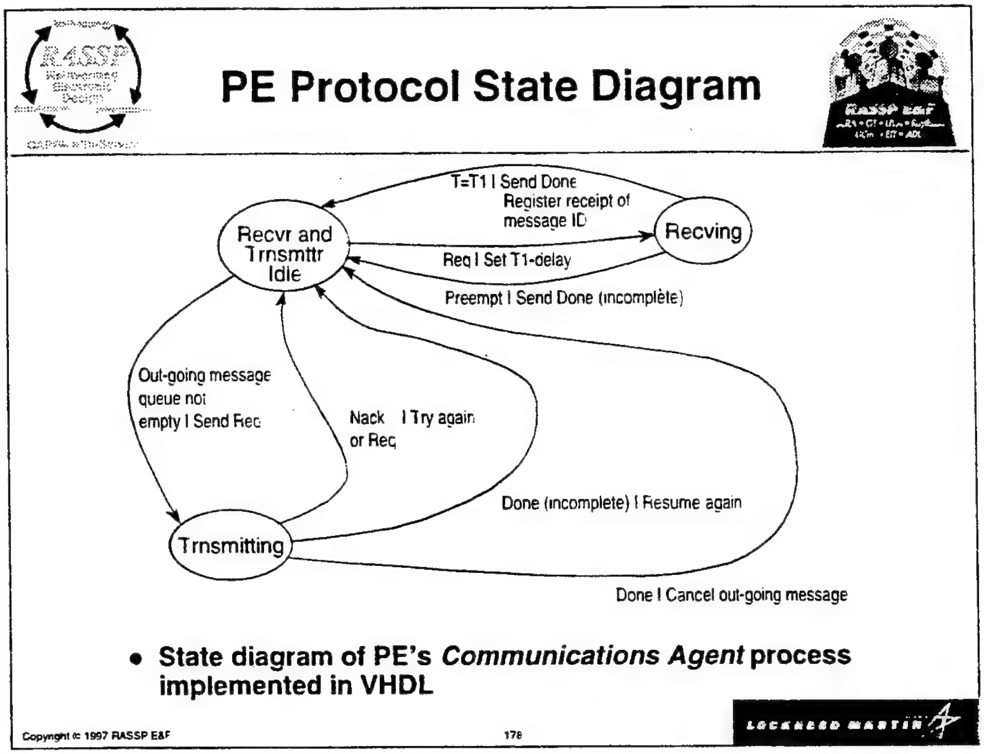
Preemption

T0₀ PE1 —REQ→ Xbar — PE2
T0₁ PE1 — Xbar —REQ→ PE2
T2₀ PE1 — Xbar —Freempt→ PE2
T2₁ PE1 — Xbar ←DONE— PE2
T2₂ PE1 ←DONE— Xbar — PE2

Contention

T0₀ PE1 —REQ→ Xbar — PE2
T0₁ PE1 ←NACK— Xbar — PE2

These figures illustrate how preemption and contention (requesting a busy path) are handled in the simplified ATL protocol.

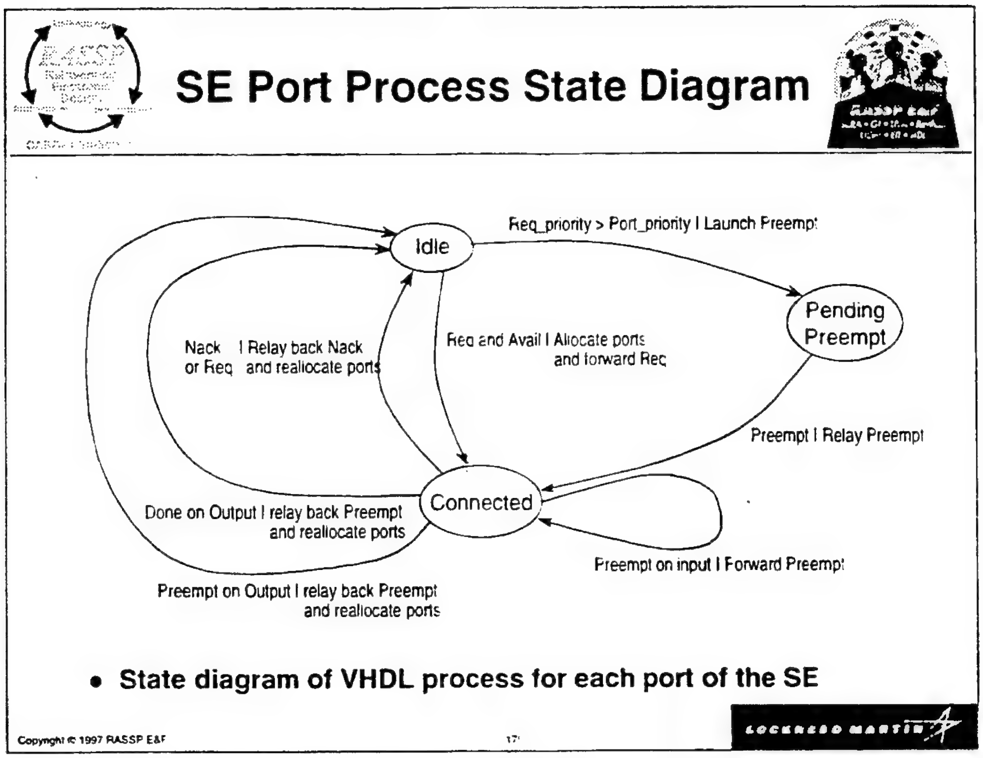


# PE Protocol State Diagram

Recvr and Trnsmttr Idle
Recving
Trnsmitting
T=T1 I Send Done Register receipt of message ID
Req I Set T1-delay
Preempt I Send Done (incomplete)
Out-going message queue not empty I Send Req
Nack I Try again or Req
Done (incomplete) I Resume again
Done I Cancel out-going message

- **State diagram of PE's *Communications Agent* process implemented in VHDL**

The communications agent and how it handles the various network functions such as requesting a path for a message, sending the message, and responding to preemption, is fairly complex, so it was designed as a state machine. This state machine was then implemented in VHDL to perform the required function. Note that within the PE VHDL code, the communications agent and computation agent pass data back and forth using shared variable instead of signals, further reducing simulation execution time.

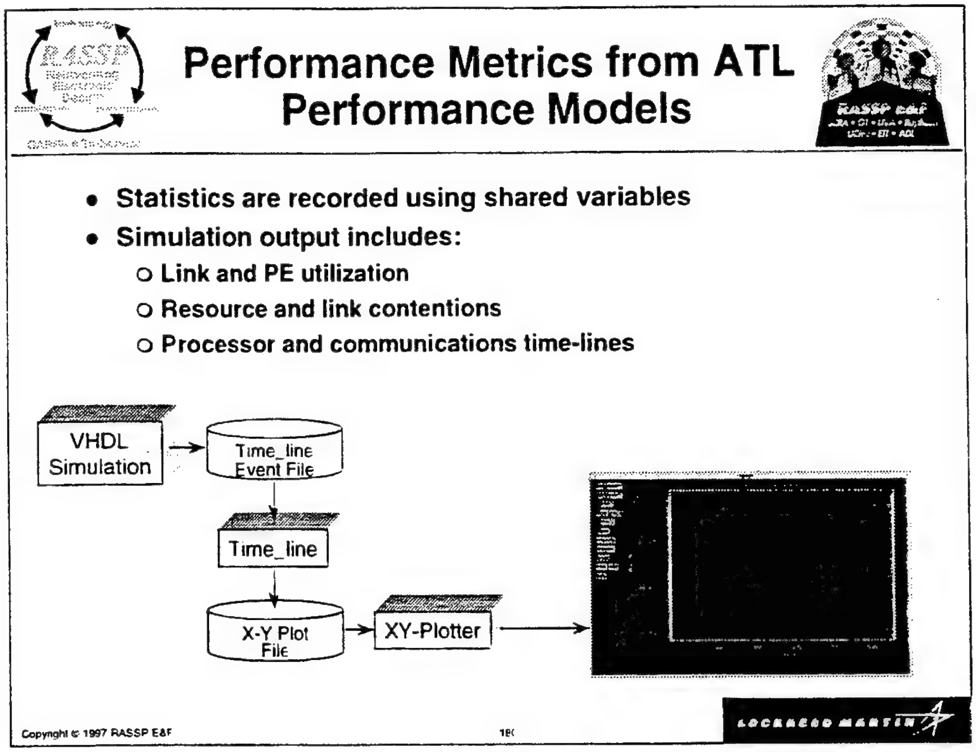


# SE Port Process State Diagram

Idle
Req_priority > Port_priority | Launch Preempt
Pending Preempt
Nack | Relay back Nack
or Req and reallocate ports
Req and Avail | Allocate ports
and forward Req
Preempt | Relay Preempt
Connected
Done on Output | relay back Preempt
and reallocate ports
Preempt on input | Forward Preempt
Preempt on Output | relay back Preempt
and reallocate ports

- **State diagram of VHDL process for each port of the SE**

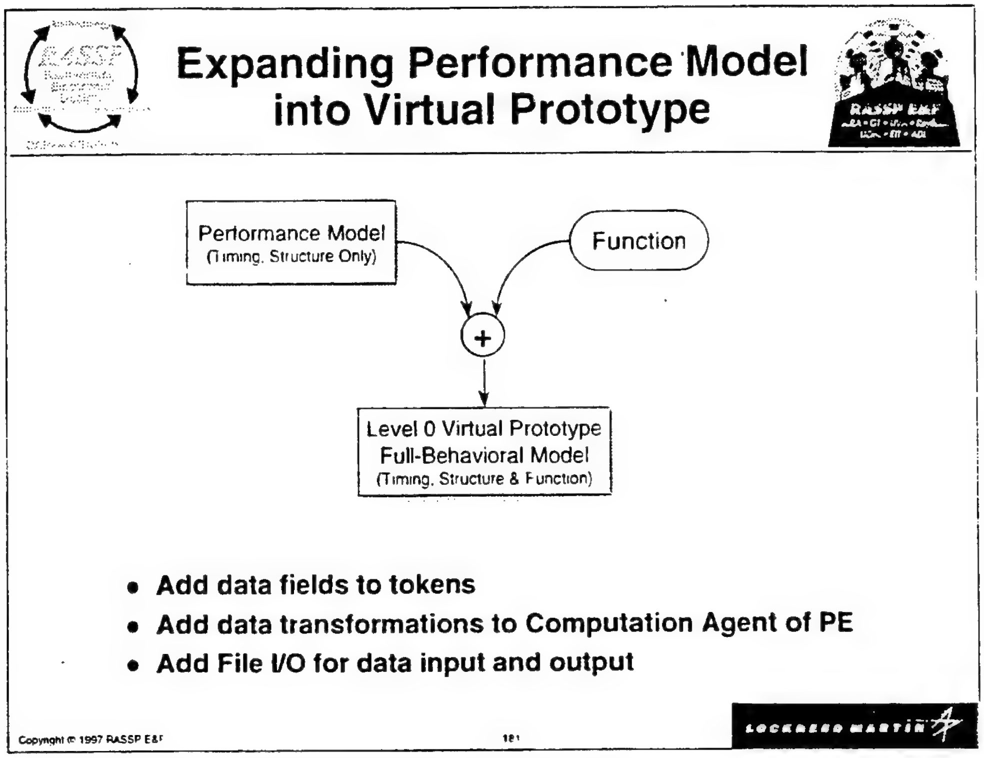
This is the state diagram for the VHDL process that implements the procedures of the port in the switch element (crossbar).



## Performance Metrics from ATL Performance Models

- Statistics are recorded using shared variables
- Simulation output includes:
  - ○ Link and PE utilization
  - ○ Resource and link contentions
  - ○ Processor and communications time-lines

A simple set of tools for collecting and analyzing performance metrics from the ATL modules was devised. The main tool is a time line utilization analysis tool that is capable of displaying both the times when the PEs are busy computing and when the communications network is busy.



## Expanding Performance Model into Virtual Prototype

Performance Model
(Timing, Structure Only)

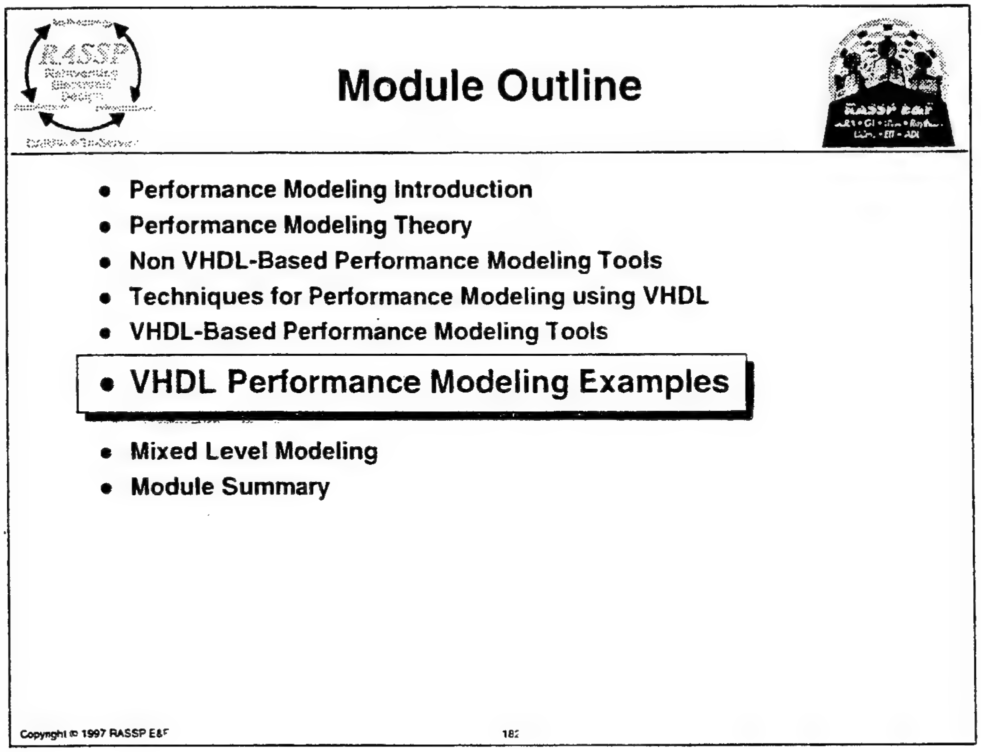
Function

+

Level 0 Virtual Prototype
Full-Behavioral Model
(Timing, Structure & Function)

- **Add data fields to tokens**
- **Add data transformations to Computation Agent of PE**
- **Add File I/O for data input and output**

After a high level performance model (with timing, but no functional information) is developed and analyzed, function can be added in terms of data values and data transformations. This forms what is termed in the Virtual Prototyping module as a level 0 virtual prototype (high level function plus timing).



## Module Outline

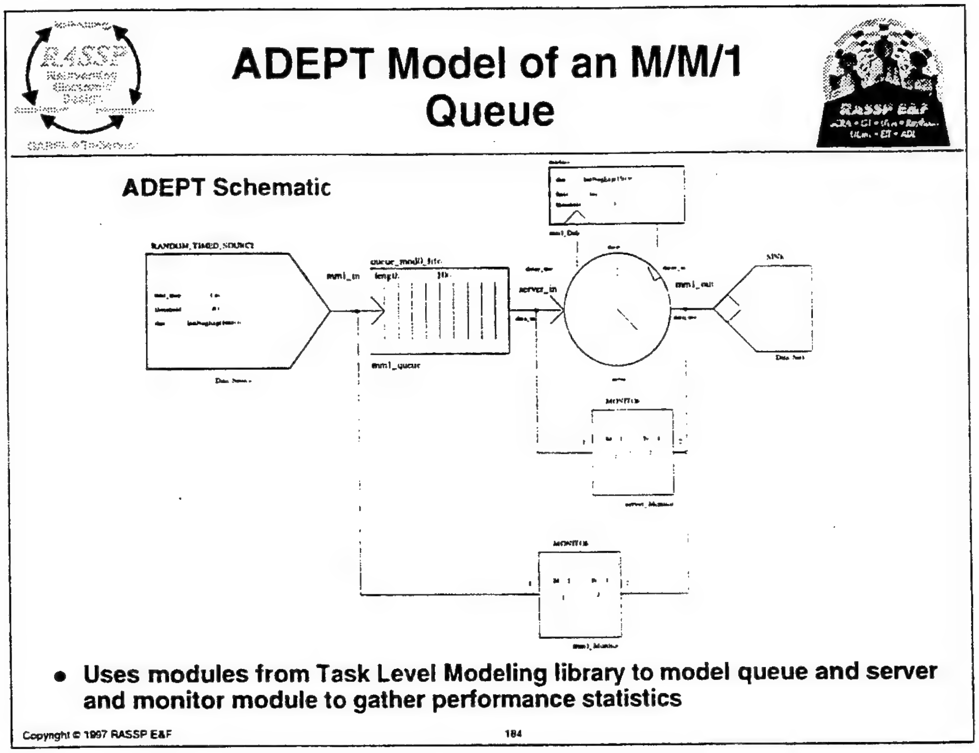
- Performance Modeling Introduction
- Performance Modeling Theory
- Non VHDL-Based Performance Modeling Tools
- Techniques for Performance Modeling using VHDL
- VHDL-Based Performance Modeling Tools
- **VHDL Performance Modeling Examples**
- Mixed Level Modeling
- Module Summary

Module Outline

# VHDL Performance Modeling Examples

- **ADEPT models of Queuing systems**
  - single M/M/1 queue
  - singe M/M/3 queue
- **High-level ADEPT model of a task graph**
  - abstract system model used to determine performance bottleneck and number of processors necessary to meet throughput requirements

There are several examples of VHDL based performance models included in this module. Most are based on the ADEPT system, but on uses the ATL performance modeling modules. However, there are many more examples available in the documentation for Cosmos and ADEPT and in the applications notes and case studies prepared as part of the RASSP program.



# ADEPT Model of an M/M/1 Queue

**ADEPT Schematic**

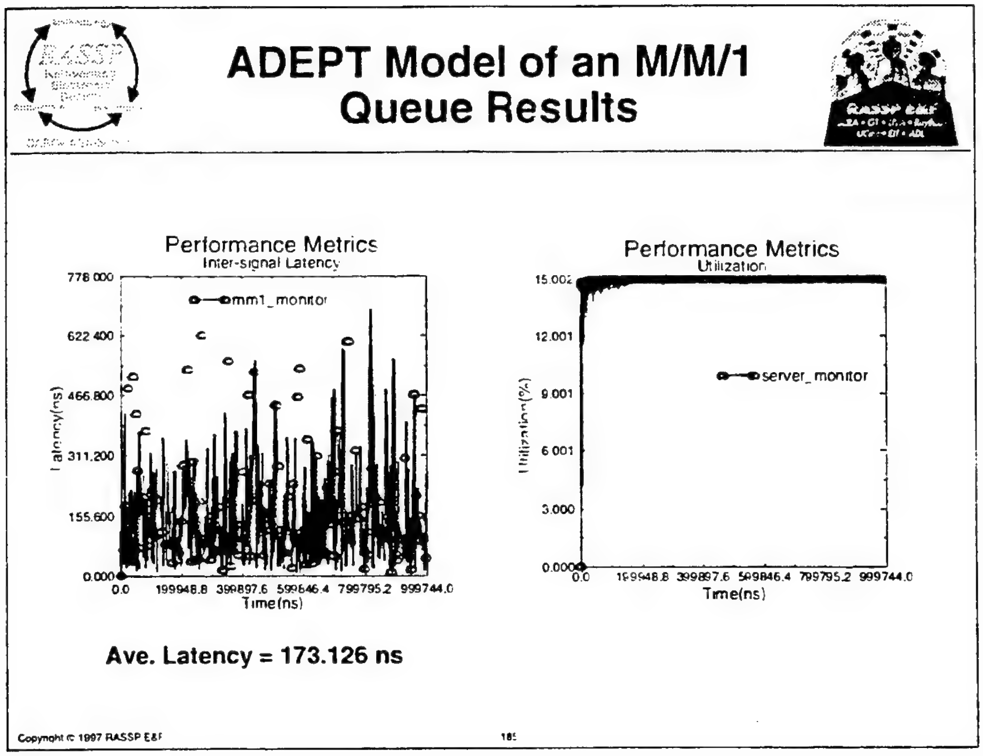
RANDOM_TIMED_SOURCE
Data Source
mm1_in
queue_mod0_fifo
length
mm1_queue
server_in
mm1_out
SINK
Data Sink
MONITOR
MONITOR

- **Uses modules from Task Level Modeling library to model queue and server and monitor module to gather performance statistics**

This is a simple model of the M/M/1 queuing system presented and analyzed earlier, using the ADEPT system. The modules used to construct this model come from the ADEPT Task Level Modeling and Module Builder's libraries.

The random_timed_source module generates a token with a random exponential arrival rate with a mean of 1000 ns (this example is modeled on a ns time scale instead of the ms time scale of the analytical example - the results are the same however). The delay module is connected to a random module such that it has a random, exponential service rate with a mean of 150 ns.

The monitor modules are standard ADEPT modules that are place in an ADEPT model to measure standard performance metrics. They record tokens as the pass by their inputs and outputs and write the information into files that are then interpreted and displayed by the ADEPT post-simulation analysis tools.

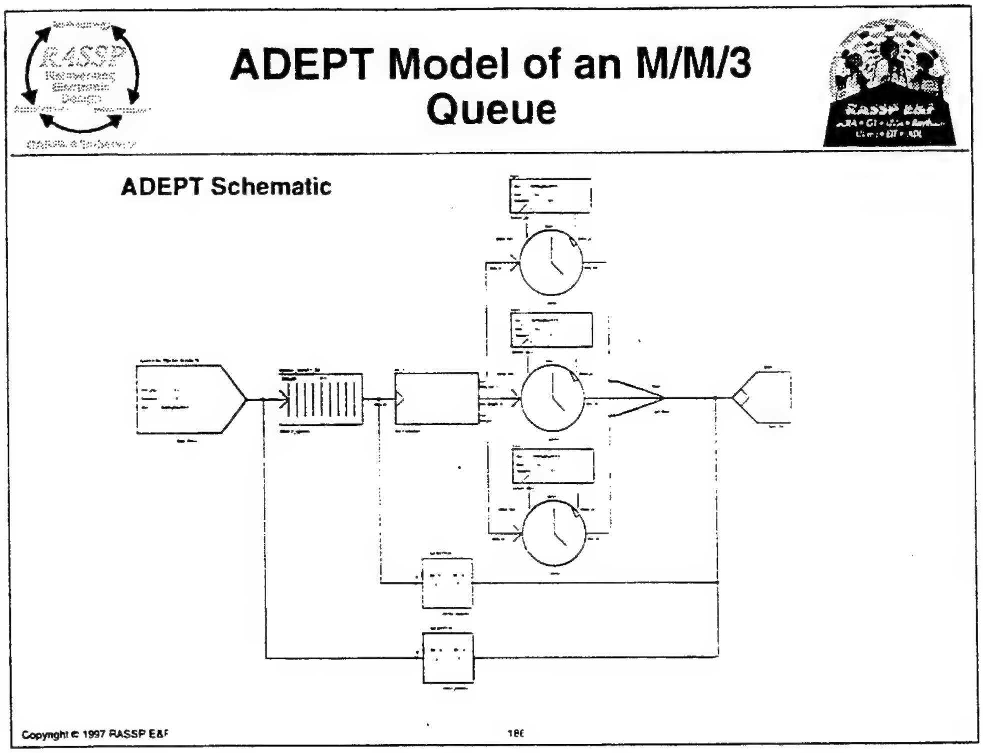


## ADEPT Model of an M/M/1 Queue Results

**Ave. Latency = 173.126 ns**

These are the results of the simulation of the M/M/1 ADEPT model. Note that the average latency of jobs (tokens) within the system is 173.126 ns as reported by the ADEPT analysis tools and that the average utilization of the server is 15%.

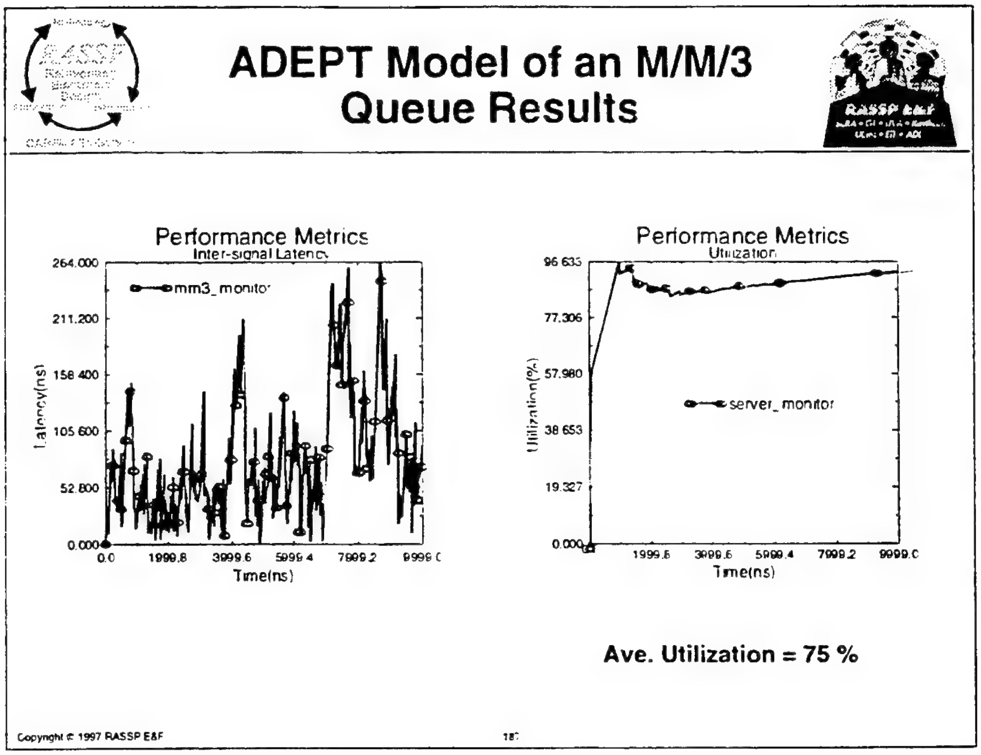
Recall that the analytical results for this model were 176.5 ns and 15% respectively.



## ADEPT Model of an M/M/3 Queue

**ADEPT Schematic**

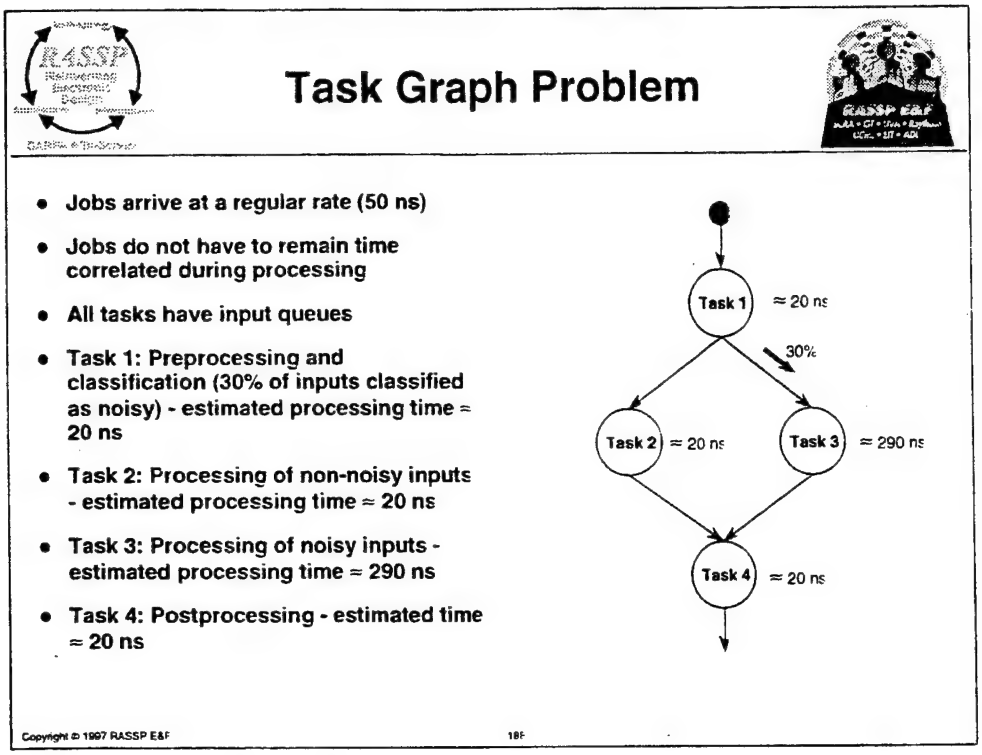
This an ADEPT model of an M/M/3 queue. It is similar to the M/M/1 model except that it obviously has three servers (delay/random module combinations). The pro_3 module is from the Task Level Modeling library and it routes tokens on its input, from the queue, to any output that is free (I.e. any server that is not busy). Note that it has a built-in priority that if more than one server is free, then it routes the token (job) to the lowest numbered output first, but that is immaterial to this model.



## ADEPT Model of an M/M/3 Queue Results

**Ave. Utilization = 75 %**

Here are the results of the ADEPT M/M/3 model. Note that the average utilization for the servers is 75% which agrees with the analytical results and the average latency seems to be close to the analytical result of 81 ns (again, this simulation was on a ns scale as opposed to the ms scale of the analytical analysis).

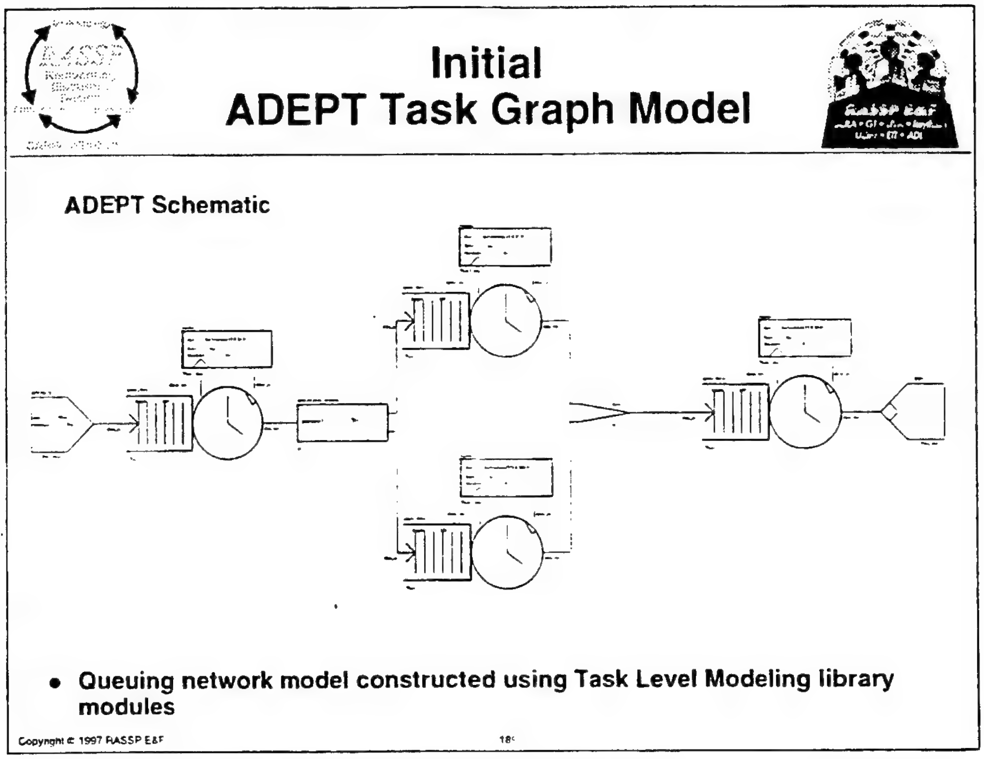


## Task Graph Problem

- **Jobs arrive at a regular rate (50 ns)**
- **Jobs do not have to remain time correlated during processing**
- **All tasks have input queues**
- **Task 1: Preprocessing and classification (30% of inputs classified as noisy) - estimated processing time ≈ 20 ns**
- **Task 2: Processing of non-noisy inputs - estimated processing time ≈ 20 ns**
- **Task 3: Processing of noisy inputs - estimated processing time ≈ 290 ns**
- **Task 4: Postprocessing - estimated time ≈ 20 ns**

This is a simple task graph problem that further illustrates the ADEPT performance modeling environment. In this problem, there is a set of jobs (say images to process) that arrive from a sensor at a regular rate. The first task is to classify the images as to their clarity - noisy or non-noisy. An average of 30% of the images are classified as noisy and must be filtered. The remaining non-noisy images must be formatted, but that takes much less time than the filtering operation. Finally, all images must be compressed for storage. Images do not need to remain correlated in the time that they arrived as they pass through the system, I.e., non-noisy images may move ahead of noisy images during processing.

An ADEPT model will be constructed to explore the issue of how many processors are required to perform the noisy image filtering to meet throughput requirements. A more detailed version of this model, with links to lower levels of hardware/software codesign and mixed level modeling, is available in the standard ADEPT deliverable.

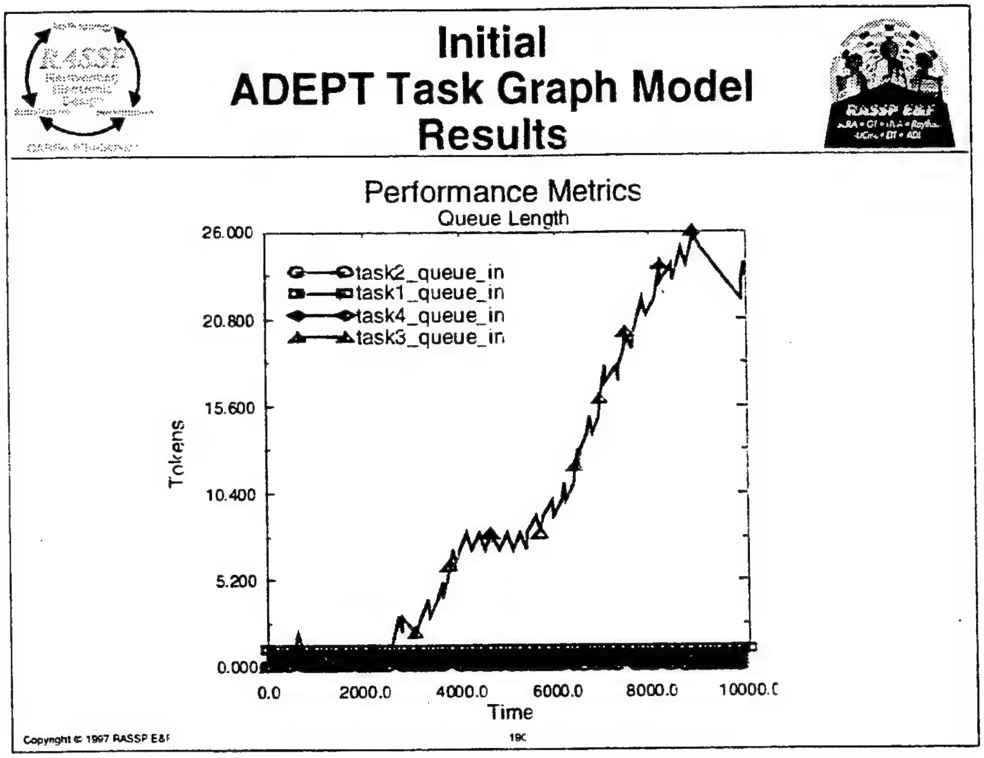


## Initial ADEPT Task Graph Model

**ADEPT Schematic**

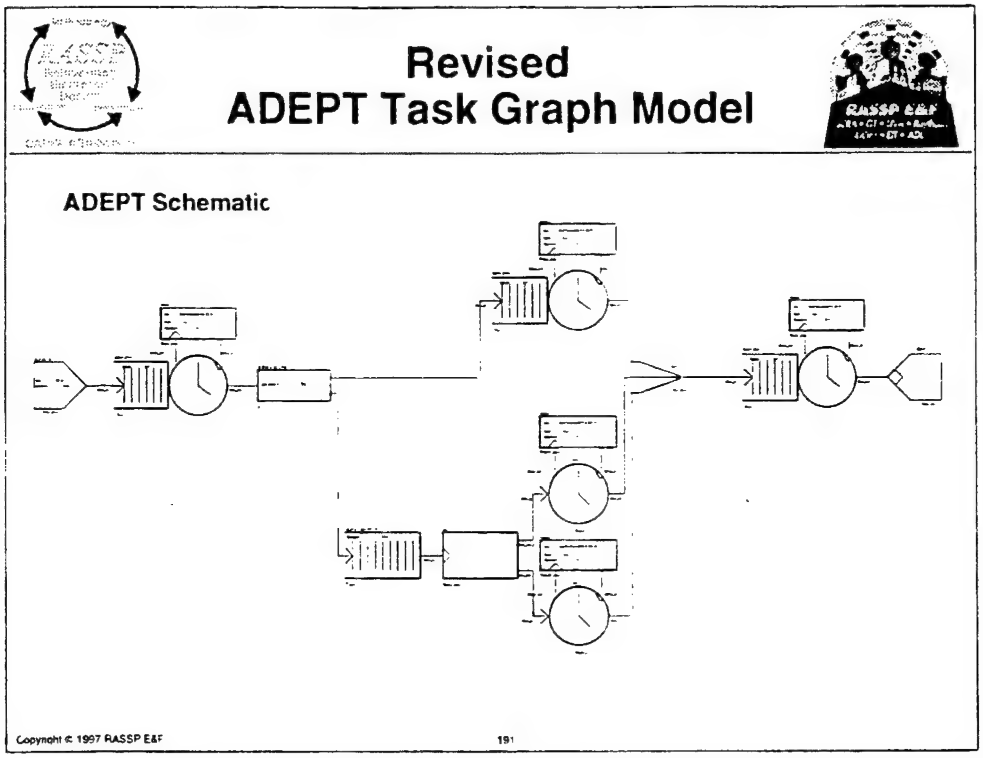
- **Queuing network model constructed using Task Level Modeling library modules**

This is the initial ADEPT performance model of the task graph problem. It is a high-level queuing network model with only one processor performing the noisy image filtering process.



# Initial ADEPT Task Graph Model Results

This is a plot of the number of items in the input queues to each task. Note that the number of items in the input queue to task 3 is increasing. Despite the slight decrease in the number of images in the queue towards the end of the simulation, it is clear that one processor is not enough to keep up with the number of filtering requests and that at least one more processor performing that task will be necessary.

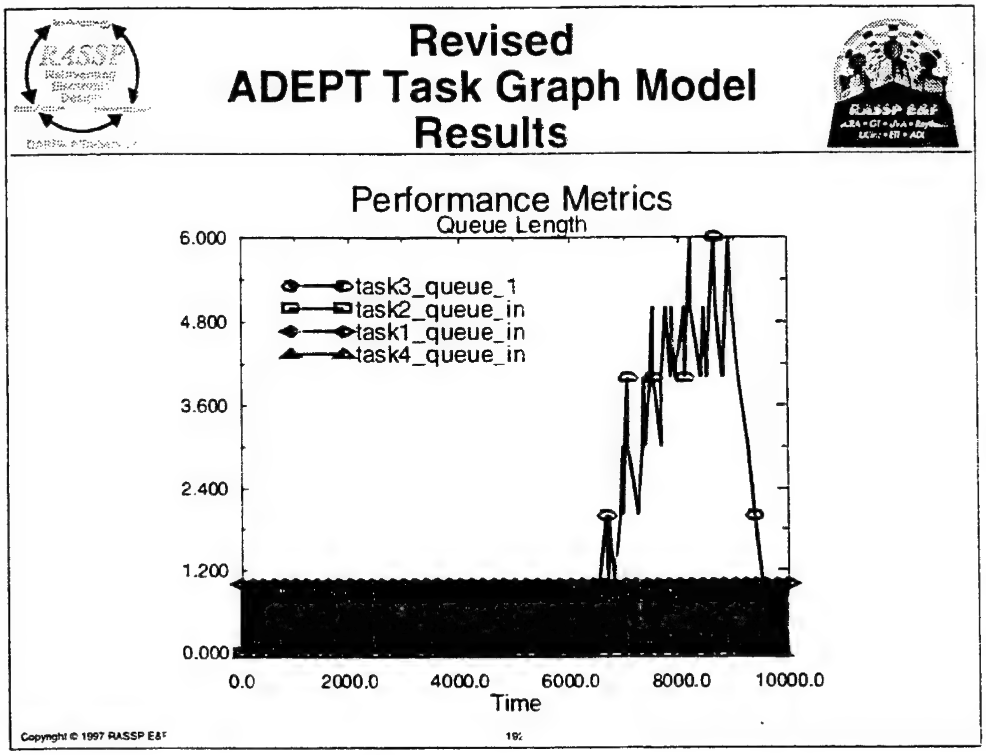


## Revised ADEPT Task Graph Model

**ADEPT Schematic**

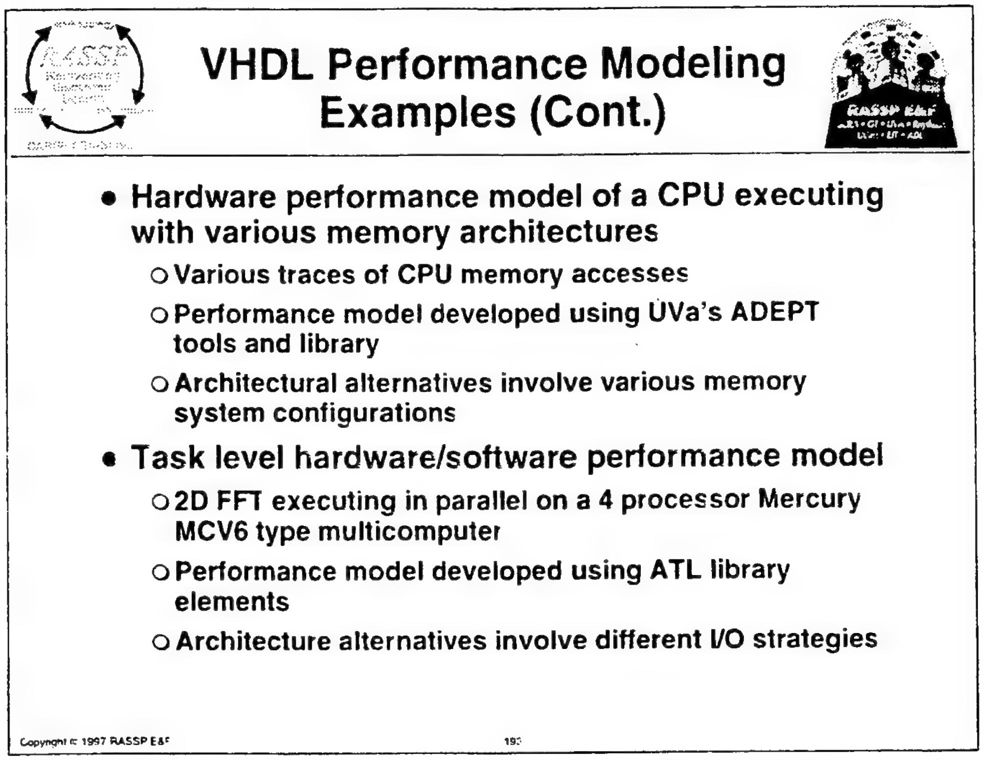
Here is a model with tow processors for task 3. Again, a pro_2 module is used to schedule jobs from the task 3 queue onto idle task 3 processors.

Again, a more detailed model of this scenario, where task 3 is taken down one more level to model actual software algorithms executing on a Digital Signal Processor, and task 4 is taken down to a behavioral model of an ALU using mixed level modeling, is included in the ADEPT package.



## Revised ADEPT Task Graph Model Results

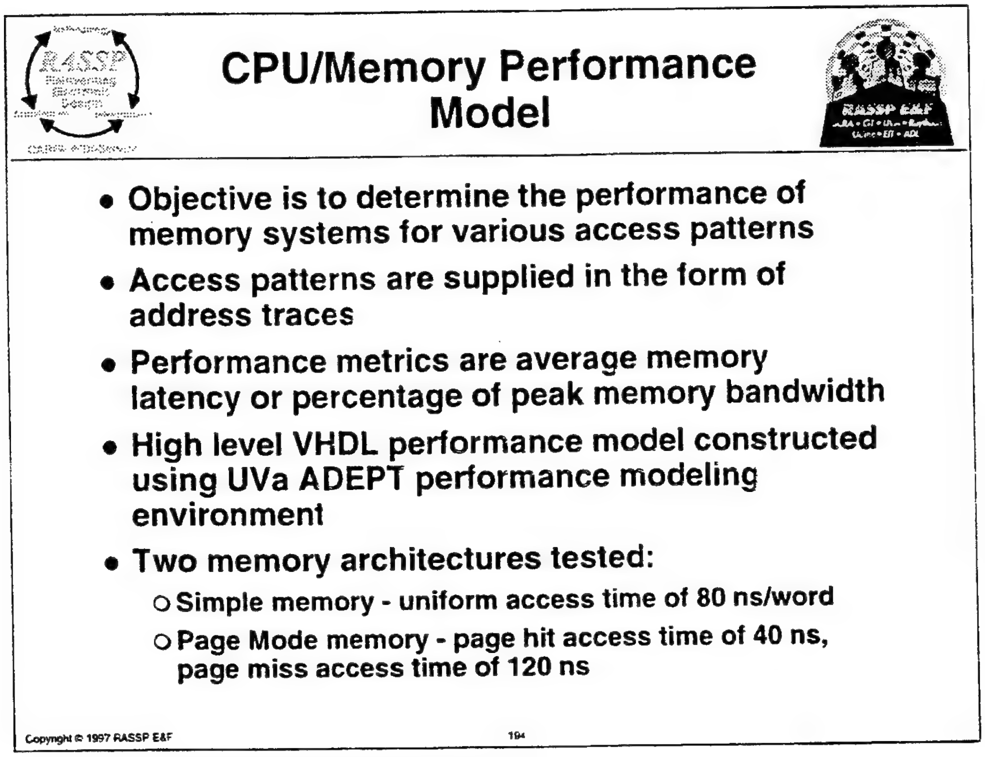
Here is the plot of queue depths for the two task 3 processor model and it shows that the depth of the task 3 queue is bounded, so two processors for that task should be enough. However, more detail should be added to the model to further prove this conclusion as the results show that the task 3 queue still may fill up if the estimate of the time required to perform the filtering is optimistic.



## VHDL Performance Modeling Examples (Cont.)

- Hardware performance model of a CPU executing with various memory architectures
  - Various traces of CPU memory accesses
  - Performance model developed using UVa's ADEPT tools and library
  - Architectural alternatives involve various memory system configurations
- Task level hardware/software performance model
  - 2D FFT executing in parallel on a 4 processor Mercury MCV6 type multicomputer
  - Performance model developed using ATL library elements
  - Architecture alternatives involve different I/O strategies

Next will be presented two more performance modeling example. One, a performance model of a CPU and memory modeled with ADEPT, and another, a hardware/software task level performance model done with the ATL performance modeling modules.



## CPU/Memory Performance Model

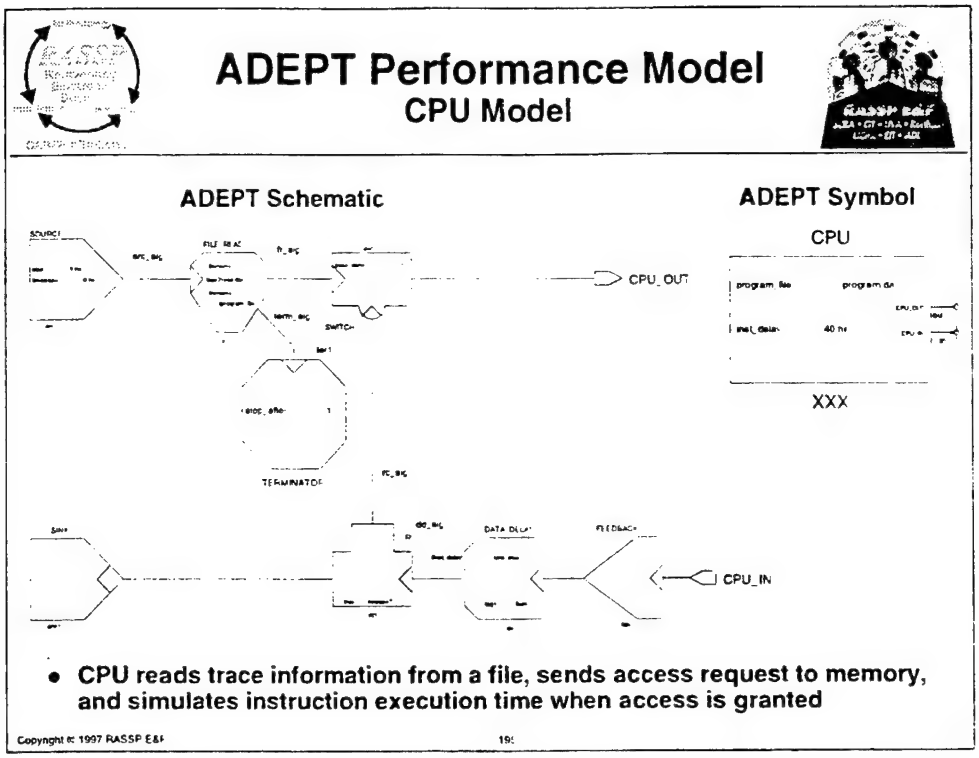
- Objective is to determine the performance of memory systems for various access patterns
- Access patterns are supplied in the form of address traces
- Performance metrics are average memory latency or percentage of peak memory bandwidth
- High level VHDL performance model constructed using UVa ADEPT performance modeling environment
- Two memory architectures tested:
  - Simple memory - uniform access time of 80 ns/word
  - Page Mode memory - page hit access time of 40 ns, page miss access time of 120 ns

The CPU/memory performance model is a simple example of a "hardware only" type of performance model. The objective of the performance model is to be able to determine the performance of various memory system architectures on typical memory traces.

At this point, only two different memory architectures were tested:

- a simple memory model in which each access takes a uniform time (based on the size of the access) of 80 ns per word.

- a page mode dram memory model where the memory system is divided up into "pages" of a specified size. If an access is made to a memory location that is on the same page as the one immediately preceding in, the "page hit access time" is 40 ns. If the access is on a different page, then the current page has to be closed and a new one opened which results in a "page miss access time" of 120 ns. Therefore, grouping accesses into groups that hit the same page (as will be seen in the DAXPY example trace) can result in significantly decreased access time.

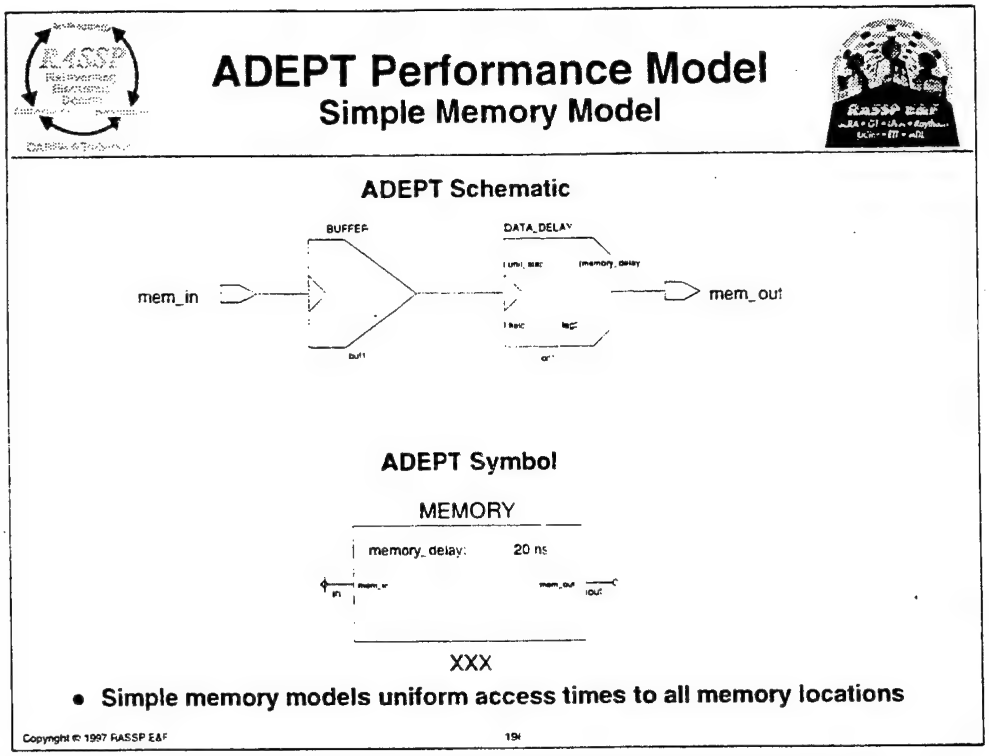


## ADEPT Performance Model

### CPU Model

- **CPU reads trace information from a file, sends access request to memory, and simulates instruction execution time when access is granted**

This is the simple CPU model. At the start of simulation time, the Source module generates a token which passes through the File_read module and picks up the first set of trace information. The token then weights at the Switch module until it is released by it. Also at time zero, the Feedback module generates an initial token (once at time 0 only) which enters the Data_delay module. The Data_delay module models the actual execution of instructions by the CPU and delays the CPU's instruction time (10 ns) times the number of instructions the current memory access allows to execute (contained on tag1 of the token). The initial token from the feedback module delays for one instruction (10 ns) and then passes through the RC module. The RC module produces a "control" token on its output which is connected to the Switch module which causes the Switch to release the next token to the memory system. The token from the RC module is then consumed by the Sink module. After the token leaves the switch module and is passed to the memory system model (through the CPU_OUT port), the Source module produces another token which passes through the File_Read module and waits at the Switch module until it is released by the token returning from the memory model (through the CPU_IN port). When the File_read module reaches the end of the address trace file, it sends a "control" token to the terminator module which terminates the simulation.



## ADEPT Performance Model

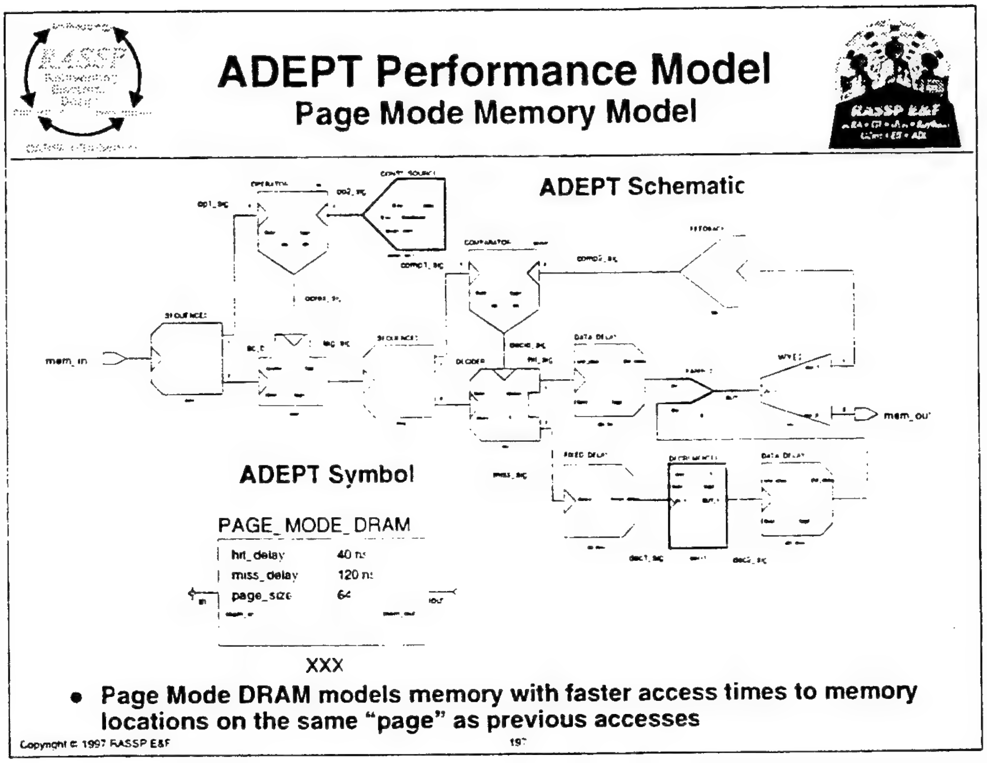
### Simple Memory Model

**ADEPT Schematic**

**ADEPT Symbol**

- **Simple memory models uniform access times to all memory locations**

This is the simple memory system model. When the token arrives from the CPU (through the MEM_IN port), it is buffered by the buffer module and then waits at the data delay module for a time determined by the number of words the access is for (determined by tag2 of the incoming token). Notice that the access time is independent of the actual address that is addresses (specified by tag3 of the token). Also note that the default delay time is 20 ns per word, but that is overwritten by the 80 ns specified on the top level schematic (as seen in the coming slide).



# ADEPT Performance Model

## Page Mode Memory Model

**ADEPT Schematic**

**ADEPT Symbol**

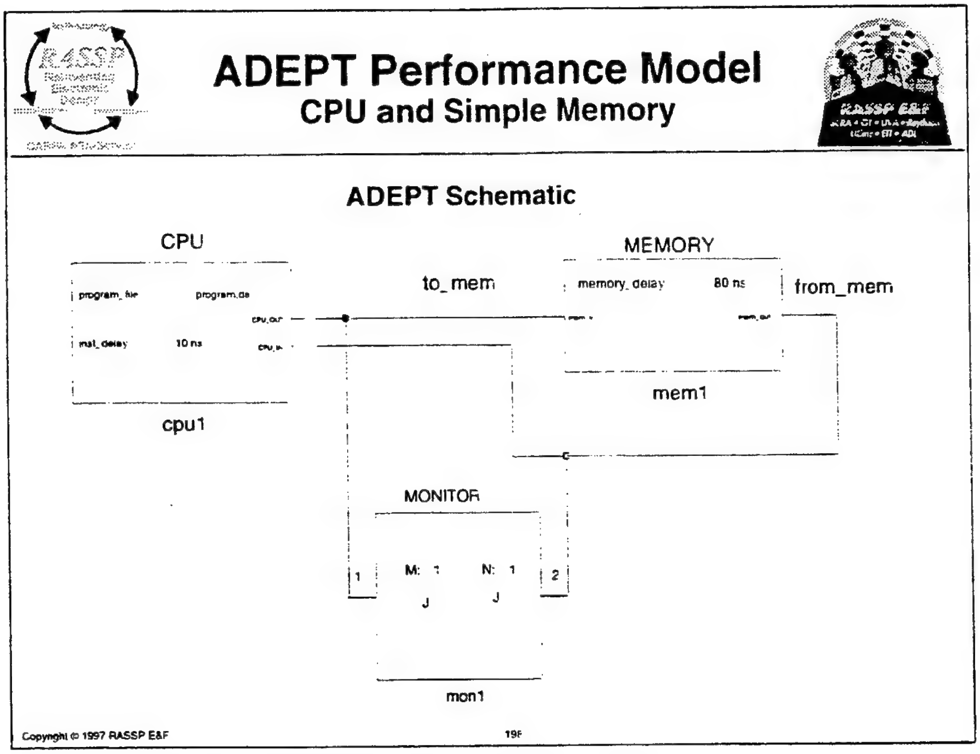
PAGE_MODE_DRAM

| | |
|---|---|
| hit_delay | 40 ns |
| miss_delay | 120 ns |
| page_size | 64 |

XXX

- **Page Mode DRAM models memory with faster access times to memory locations on the same "page" as previous accesses**

This is the model of the page mode dram which is more complex than the simple memory model, but still very straight froward. When a token enters the model (through the MEM_IN port), the Sequence module creates a copy of it and send it to the Operator module. The address of that token (on tag3) is divided by the specified page size (provided on tag3 of the other token input to the Operator by the Constant Source module and the Page_size generic on the overall symbol) to generate the resulting page number on tag1 of the output at the bottom of the Operator module. Once this process is complete, the first Sequence module passes the original token to the SC_D module where the page number is written onto tag4 for the token. It then passes to the second Sequence module which creates a copy of the token and send it to the Comparator module. The comparator module compares the page number on tag4 of the token to the previous page number stored on its other input token. If they are equal, the Comparator signals the Decider module to send the original token through the Data_ delay that has the hit_delay. If they are not equal, the Decider sends the token though the lower path. In the lower path, the token is delayed for one miss_delay time to simulate the opening of the new page and the accessing of the first word of the request. Then the number of words requested is decremented by one and the token is delayed for the remaining number of words times the hit_delay. Finally the token passes through the Wye module which sends one copy of the token, containing the new current page number on its tag4, to the Comparator module and another copy out of the memory back to the CPU.

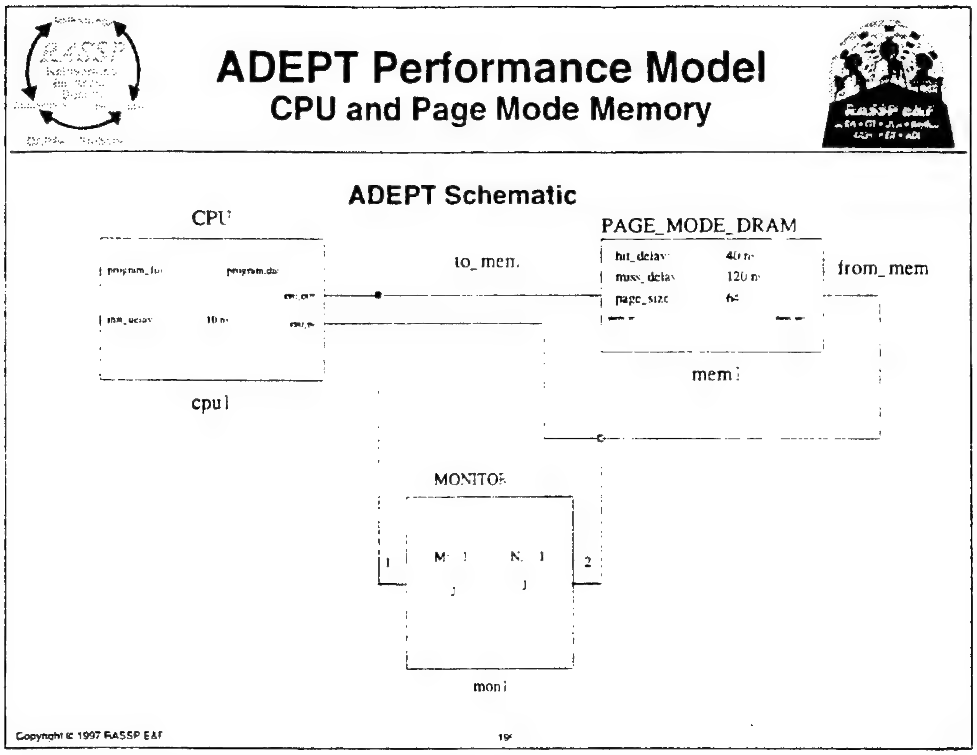


# ADEPT Performance Model

## CPU and Simple Memory

### ADEPT Schematic

This is the ADEPT schematic of the overall model with the simple memory. Notice that the memory access time on the memory model has been changed to 80 ns which will override the 20 ns default as explained before.

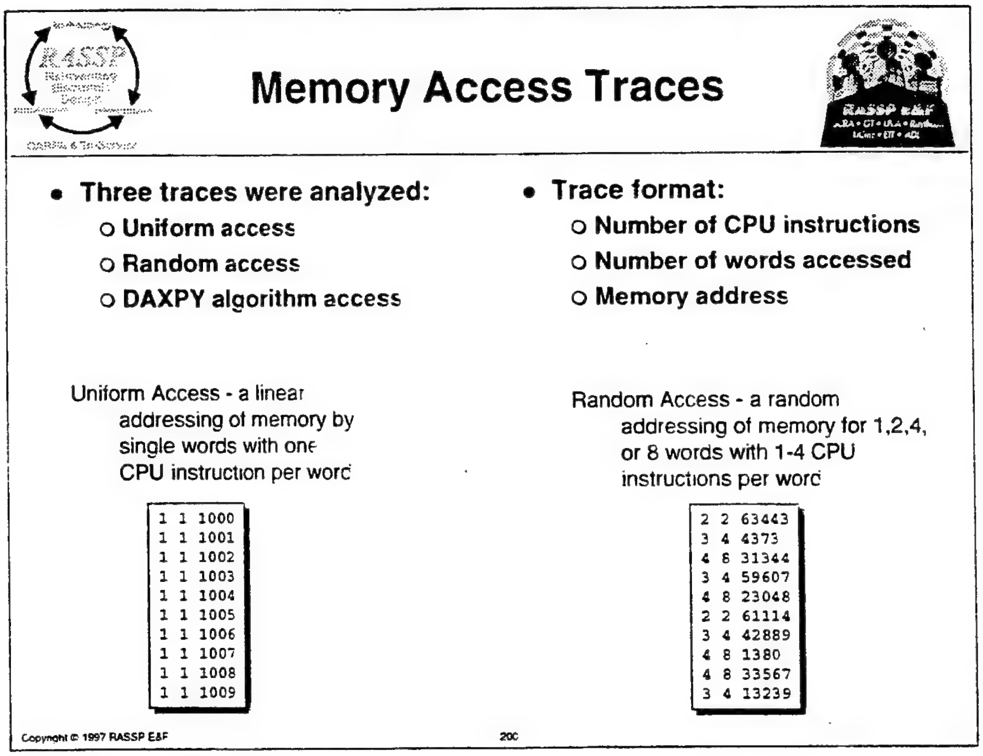


## ADEPT Performance Model

### CPU and Page Mode Memory

**ADEPT Schematic**

This is the ADEPT schematic of the CPU with the page mode memory model.



## Memory Access Traces

- **Three traces were analyzed:**
  - o **Uniform access**
  - o **Random access**
  - o **DAXPY algorithm access**
- **Trace format:**
  - o **Number of CPU instructions**
  - o **Number of words accessed**
  - o **Memory address**

Uniform Access - a linear addressing of memory by single words with one CPU instruction per word

```
1 1 1000
1 1 1001
1 1 1002
1 1 1003
1 1 1004
1 1 1005
1 1 1006
1 1 1007
1 1 1008
1 1 1009
```

Random Access - a random addressing of memory for 1,2,4, or 8 words with 1-4 CPU instructions per word

```
2 2 63443
3 4 4373
4 8 31344
3 4 59607
4 8 23048
2 2 61114
3 4 42889
4 8 1380
4 8 33567
3 4 13239
```

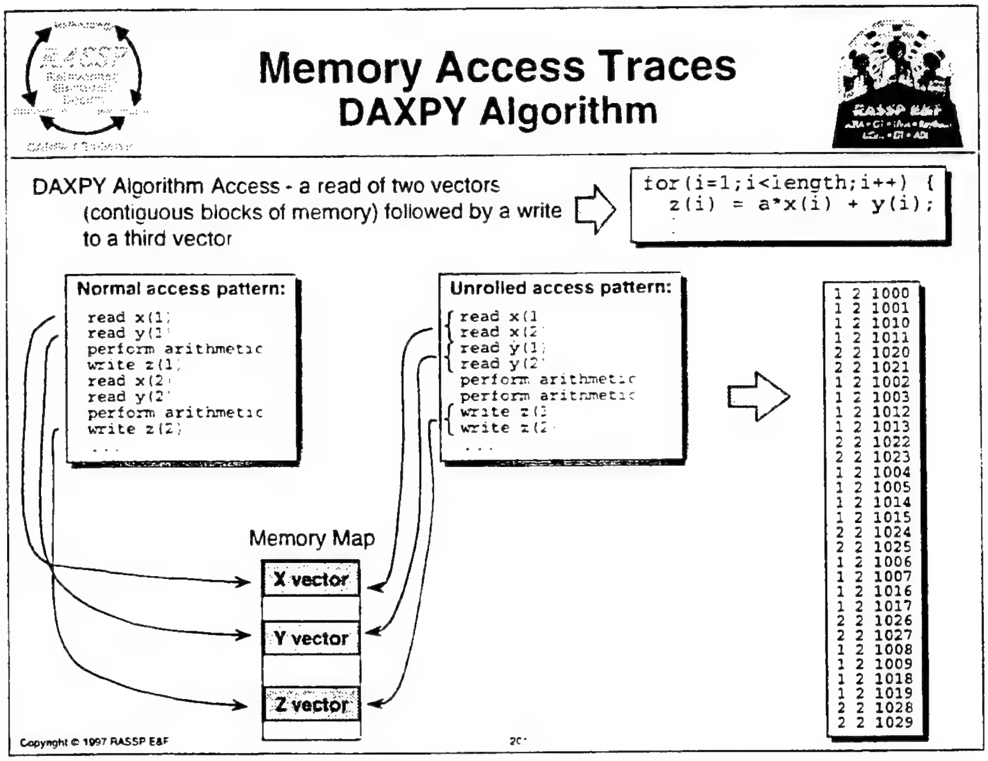
Three traces were run through the two memory system models, a simple uniform access, a random access, and a DAXPY algorithm access with loop unrolling. The traces were in the following format:

<number of instructions> <number of words> <memory address>

where number of instructions is the number of CPU instruction (time 10 ns) that the CPU will delay for after the access is granted, number of words is the number (times the access time) that the memory will delay in returning the access, and memory address is just that.

The uniform access is a single instruction, single word access where the address starts at a specify point (1000 in this example) and increments by 1 for each successive access.

The random access is an access where the number of instruction is random uniformly distribute between 1 and 4, the number of words is random uniformly distributed over the values of 1,2,4, and 8, and the address is a uniform randomly distributed number.



## Memory Access Traces
## DAXPY Algorithm

DAXPY Algorithm Access - a read of two vectors (contiguous blocks of memory) followed by a write to a third vector

```
for(i=1;i<length;i++) {
  z(i) = a*x(i) + y(i);
```

**Normal access pattern:**

```
read x(1)
read y(1)
perform arithmetic
write z(1)
read x(2)
read y(2)
perform arithmetic
write z(2)
...
```

**Unrolled access pattern:**

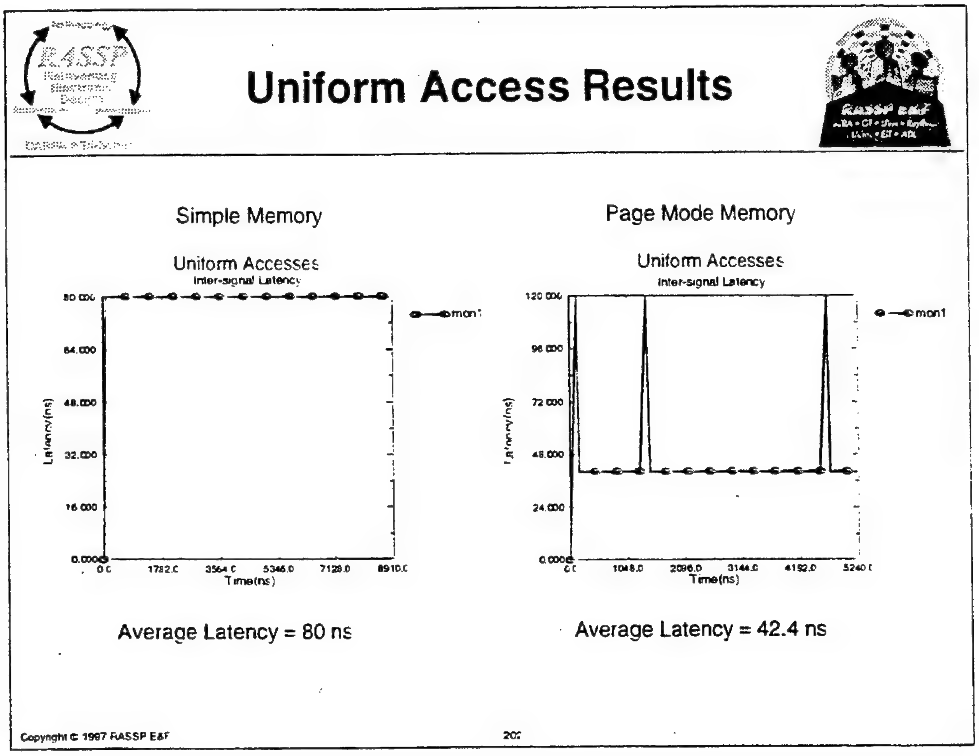
```
read x(1)
read x(2)
read y(1)
read y(2)
perform arithmetic
perform arithmetic
write z(1)
write z(2)
...
```

```
1 2 1000
1 2 1001
1 2 1010
1 2 1011
2 2 1020
2 2 1021
1 2 1002
1 2 1003
1 2 1012
1 2 1013
2 2 1022
2 2 1023
1 2 1004
1 2 1005
1 2 1014
1 2 1015
2 2 1024
2 2 1025
1 2 1006
1 2 1007
1 2 1016
1 2 1017
2 2 1026
2 2 1027
1 2 1008
1 2 1009
1 2 1018
1 2 1019
2 2 1028
2 2 1029
```

The DAXPY algorithm access is the simulation of the accesses that would happen if the CPU was running the algorithm to add two vectors (one times a constant), resulting in a third vector as shown. The vectors are stored in contiguous areas of memory as arrays that are typically on different memory pages.

If the DAXPY algorithm is executed in its native form, it will result in the pattern: read first X value, read first Y value, write first Z value, read second X value, etc. The problem with this is that if the vectors are indeed on different pages, each memory access will result in a page miss.

One solution to this is to "unroll" the loop so as to group accesses to the same page together. For example, in a twice unrolled case (loop unrolling factor of 2) the access pattern would be: read first X (and store in register) read second X, read first Y, read second Y, perform two multiply/adds, write first Z, write second Z. In this case, the first read or write would be a page miss, and the second would be a page hit. Obviously, the ideal would be to unroll the loop many times, but in reality, the amount of unrolling that can be done is limited by the size of the processor's register file.



## Uniform Access Results

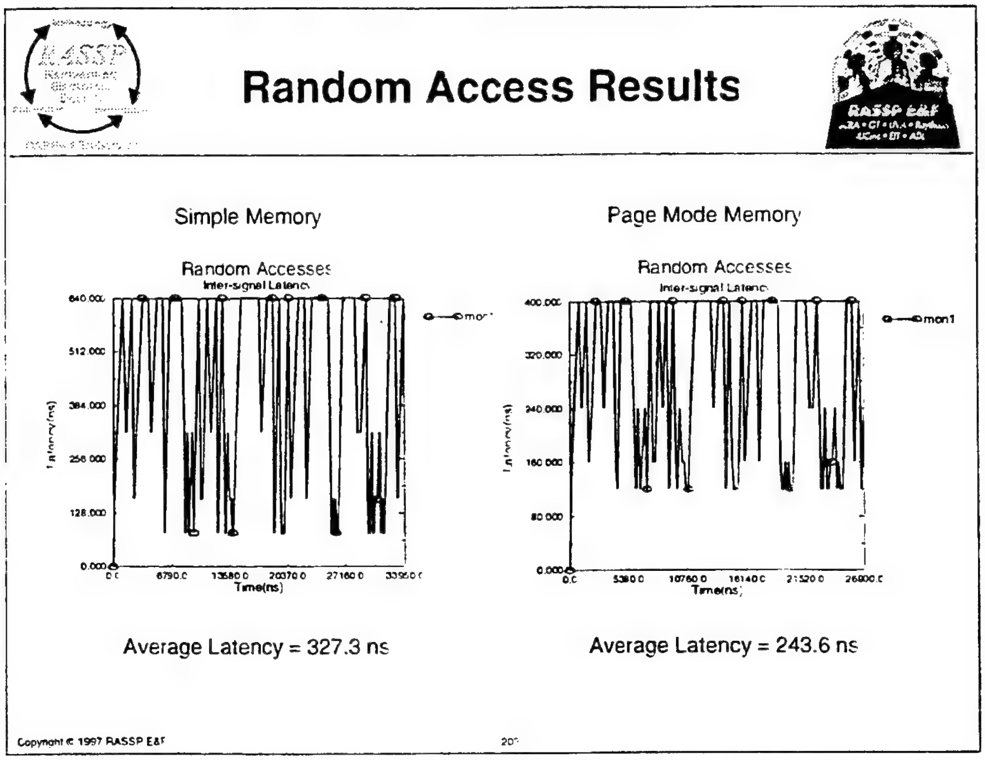
Simple Memory

Page Mode Memory

Average Latency = 80 ns

Average Latency = 42.4 ns

Here is the results for the uniform access trace for both the simple memory and the page mode DRAM. The simple memory has a uniform access time of 80 ns for each request (of one word size). The page mode DRAM has an initial access time of 120 ns, but then subsequent accesses have times of only 40 ns until the address jumps to the next page. Note that the pages in this example are 64 words long and the addresses start in the middle of a page, that's why the second miss comes earlier than the third.



# Random Access Results

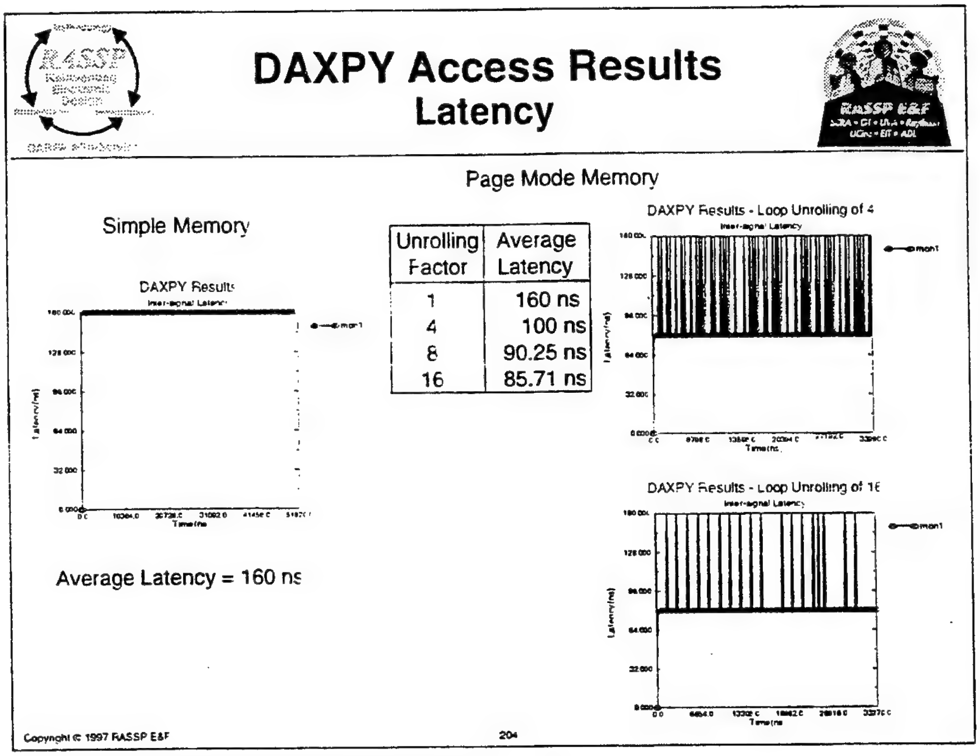
Simple Memory

Page Mode Memory

Average Latency = 327.3 ns

Average Latency = 243.6 ns

This is the results for the random access traces. The page mode DRAM is somewhat better than the simple memory here in spite of the fact that the addresses are random because many of the accesses are for multiple words and the page mode DRAM has a lower overall access time for them.



## DAXPY Access Results

### Latency

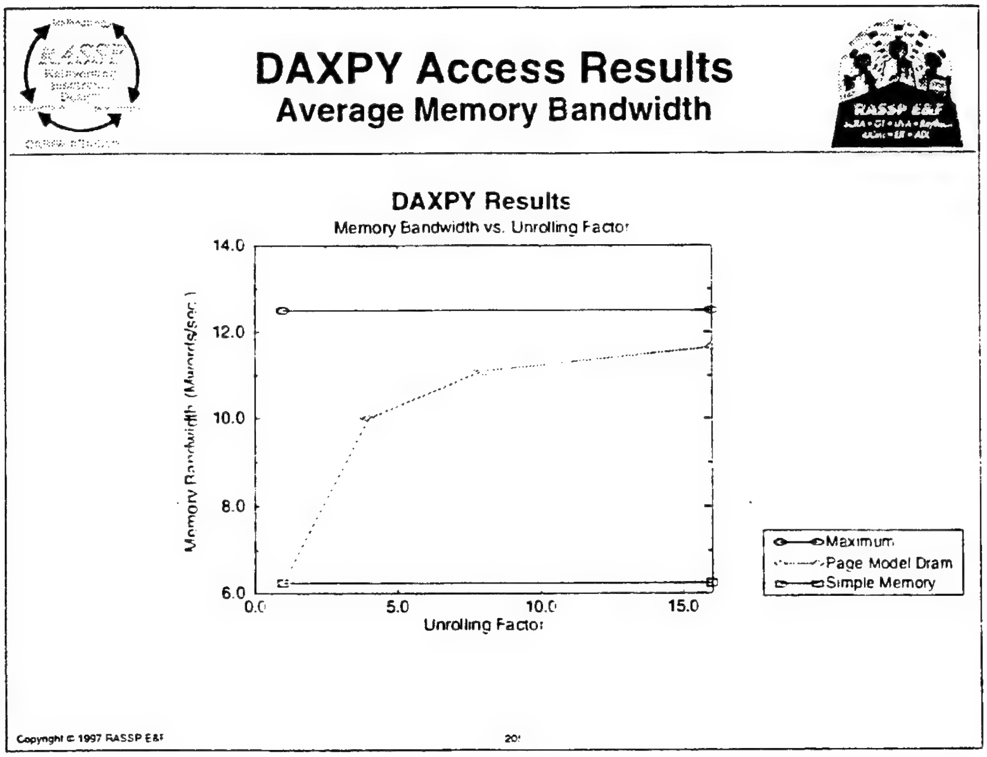
Page Mode Memory

Simple Memory

| Unrolling Factor | Average Latency |
|---|---|
| 1 | 160 ns |
| 4 | 100 ns |
| 8 | 90.25 ns |
| 16 | 85.71 ns |

Average Latency = 160 ns

Here are the results for the DAXPY accesses. The time for the simple memory is fixed because the access size is fixed. However, for the page mode DRAM, the results vary with the unrolling factor - more unrolling, lower average latency as the page misses are amortized over more page hits.

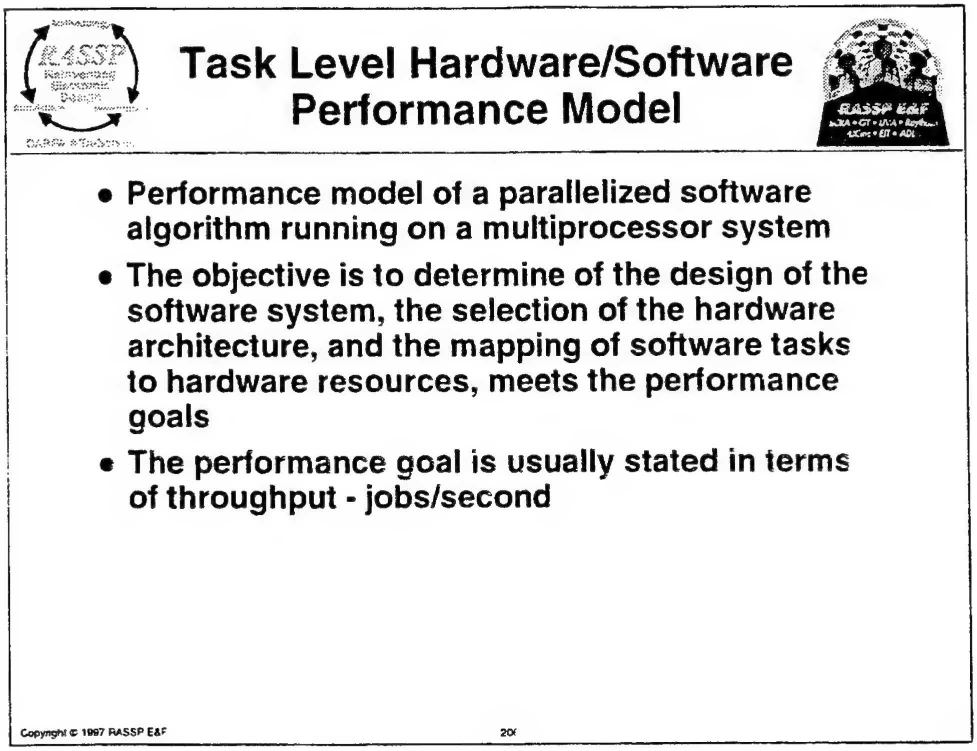


## DAXPY Access Results

### Average Memory Bandwidth

20

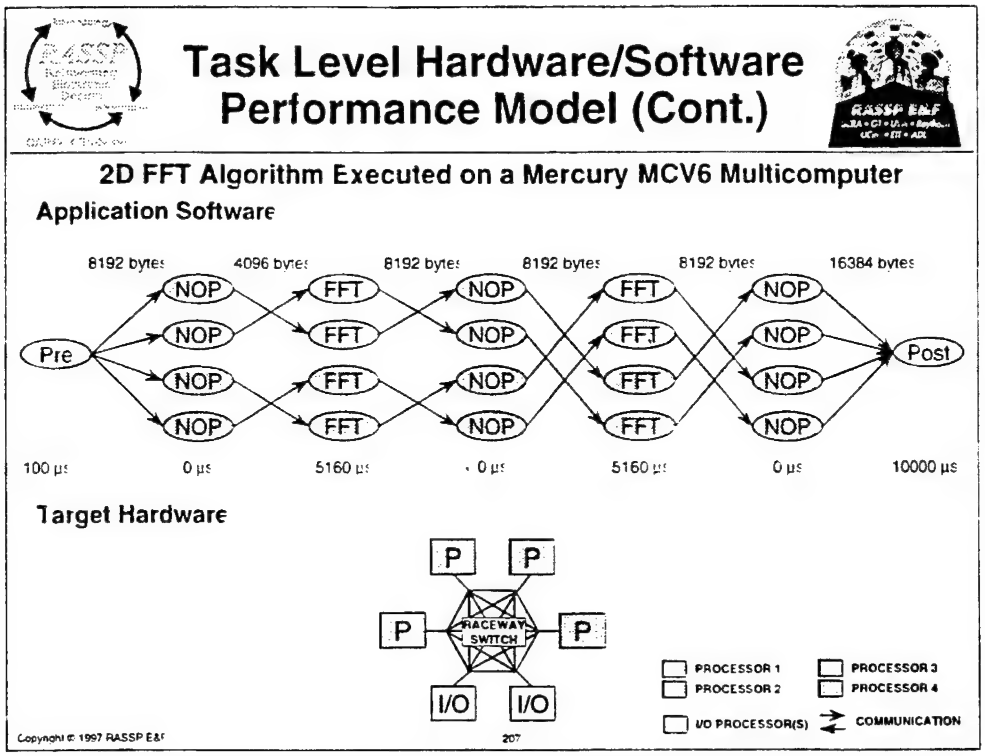
Here are the results graphed as memory bandwidth (1/average latency). Note that as the unrolling factor goes up, the average latency for the page mode DRAM approaches the theoretical maximum (1/page hit time).



## Task Level Hardware/Software Performance Model

- Performance model of a parallelized software algorithm running on a multiprocessor system
- The objective is to determine of the design of the software system, the selection of the hardware architecture, and the mapping of software tasks to hardware resources, meets the performance goals
- The performance goal is usually stated in terms of throughput - jobs/second

This section describes an example of a hardware/software performance model constructed using the ATL model elements.



## Task Level Hardware/Software Performance Model (Cont.)

**2D FFT Algorithm Executed on a Mercury MCV6 Multicomputer**

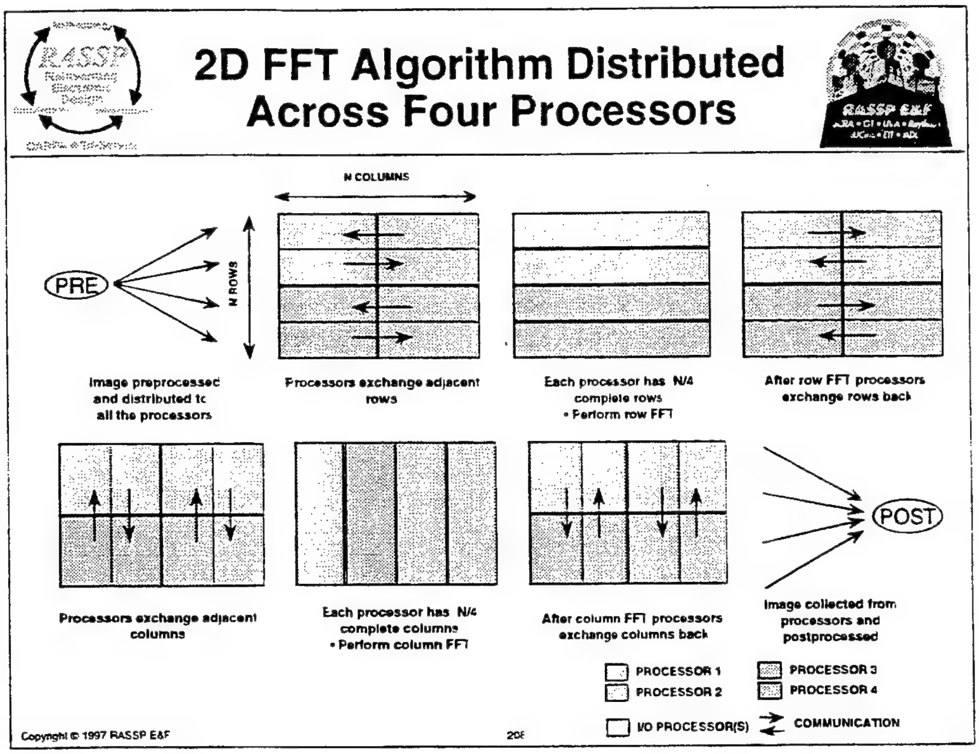
**Application Software**

**Target Hardware**

207

The upper part of the figure shows the overall structure of the software algorithm in terms of tasks, how long they require for computation (on the bottom in blue), and communications between them and the amounts of communication (in black above). The algorithm is a 2D Fast Fourier Transform (FFT). The NOPs in the algorithm are simply place holders to make the figure more clear. For example, after receiving the initial data from the pre-processing task, all of the processors, without doing any computation, exchange data with each other to perform the row FFT. This is shown in more detail on the next page.

The lower part of the figure show the hardware architecture. A 4 processor Mercury Race Multicomputer (called an MCV6), with either one or two I/O processors.



# 2D FFT Algorithm Distributed Across Four Processors

Image preprocessed and distributed to all the processors

Processors exchange adjacent rows

Each processor has N/4 complete rows
- Perform row FFT

After row FFT processors exchange rows back

Processors exchange adjacent columns

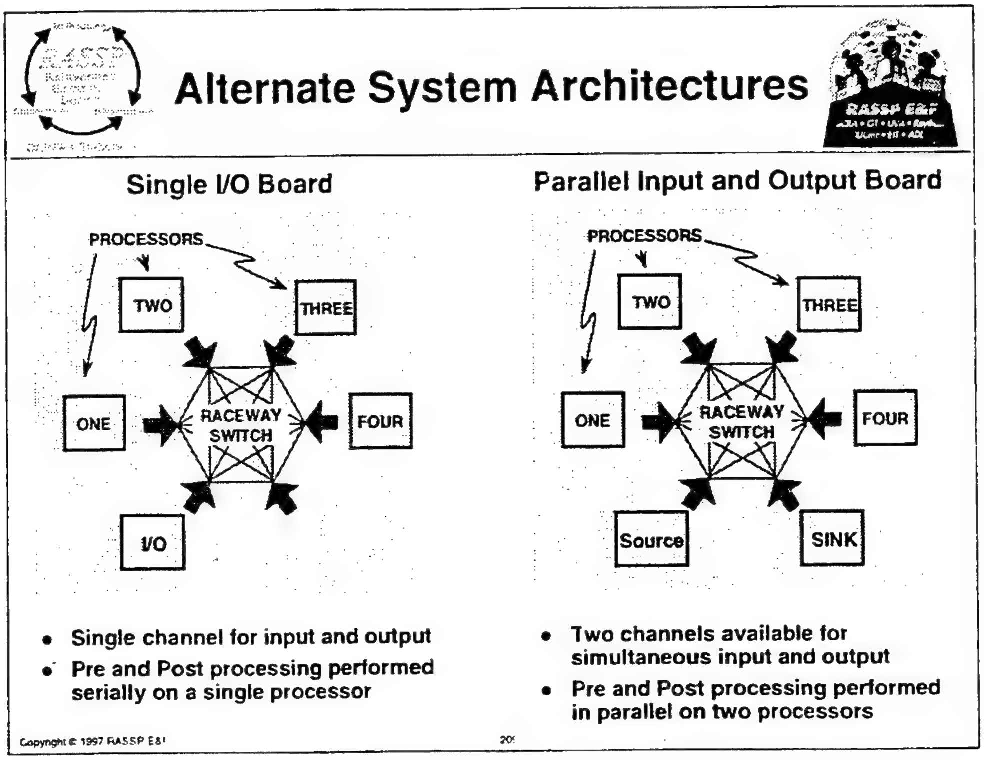
Each processor has N/4 complete columns
- Perform column FFT

After column FFT processors exchange columns back

Image collected from processors and postprocessed

☐ PROCESSOR 1
☐ PROCESSOR 2
▨ PROCESSOR 3
▨ PROCESSOR 4
☐ I/O PROCESSOR(S)
⇄ COMMUNICATION

This is more detail on how the image data is allocated to the processors and how it is exchanged during the processing of the algorithm.



## Alternate System Architectures

**Single I/O Board**

**Parallel Input and Output Board**

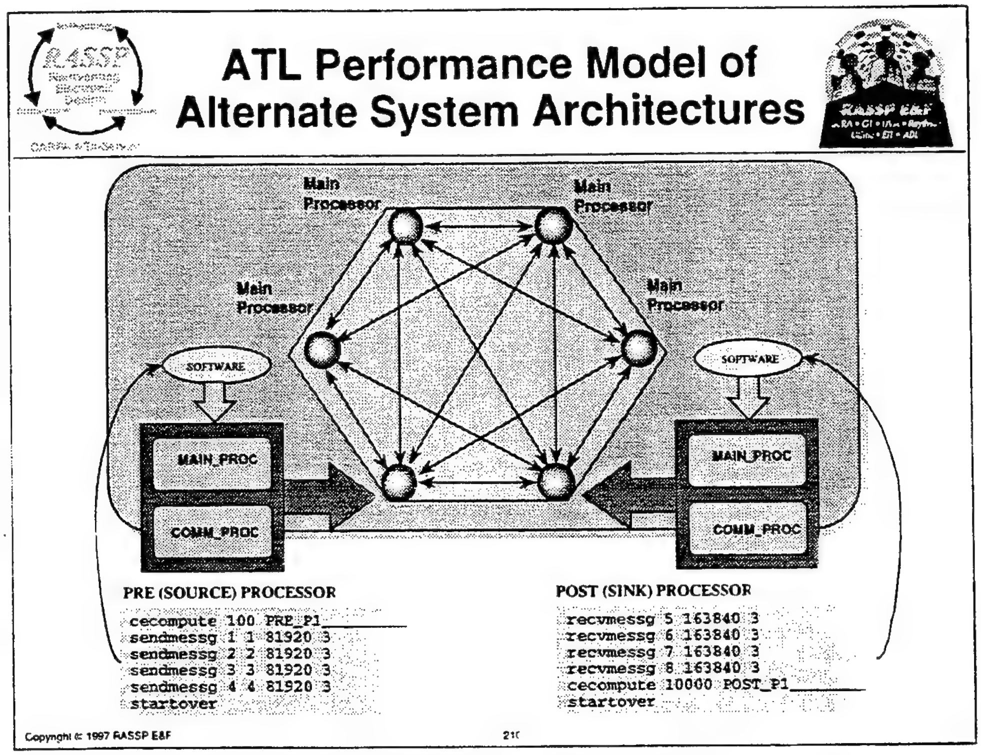
- Single channel for input and output
- Pre and Post processing performed serially on a single processor

- Two channels available for simultaneous input and output
- Pre and Post processing performed in parallel on two processors

These are the two alternate systems architectures that are investigated using the performance model. Both architectures have 4 processors and a Raceway crossbar switch, but the first architecture has a single I/O board which must perform both the pre and post-processing tasks and sending and receiving images to/from the other processors must be serialized.

In the second architecture, there is a separate source and sink processor to perform the pre and post-processing task respectively, and sending and receiving images to/from the 4 processors can occur in parallel.

Note that in the ATL performance model, regular processing elements (PEs) are used to model the I/O, Source and Sink processors.



## ATL Performance Model of Alternate System Architectures

**PRE (SOURCE) PROCESSOR**

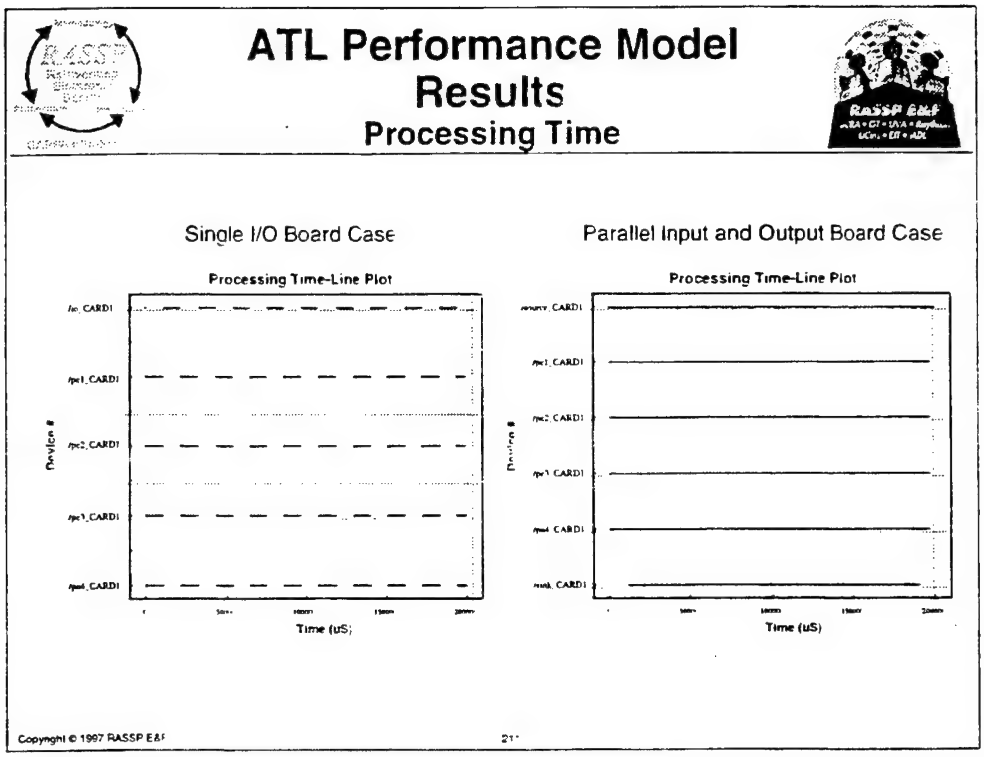
```
cecompute 100 PRE_P1_________
sendmessg 1 1 81920 3
sendmessg 2 2 81920 3
sendmessg 3 3 81920 3
sendmessg 4 4 81920 3
startover
```

**POST (SINK) PROCESSOR**

```
recvmessg 5 163840 3
recvmessg 6 163840 3
recvmessg 7 163840 3
recvmessg 8 163840 3
cecompute 10000 POST_P1________
startover
```

This slide shows more detail on the ATL performance model and how the PE modules (for the Source and Sink modules) read their programs out of a file.

Notice that the programs end in a “startover” command which makes them run the program in an endless loop. This way, the performance model can be simulated for some fixed amount of time and the number of loops which the model executes can be observed as a performance measure.



# ATL Performance Model Results

## Processing Time

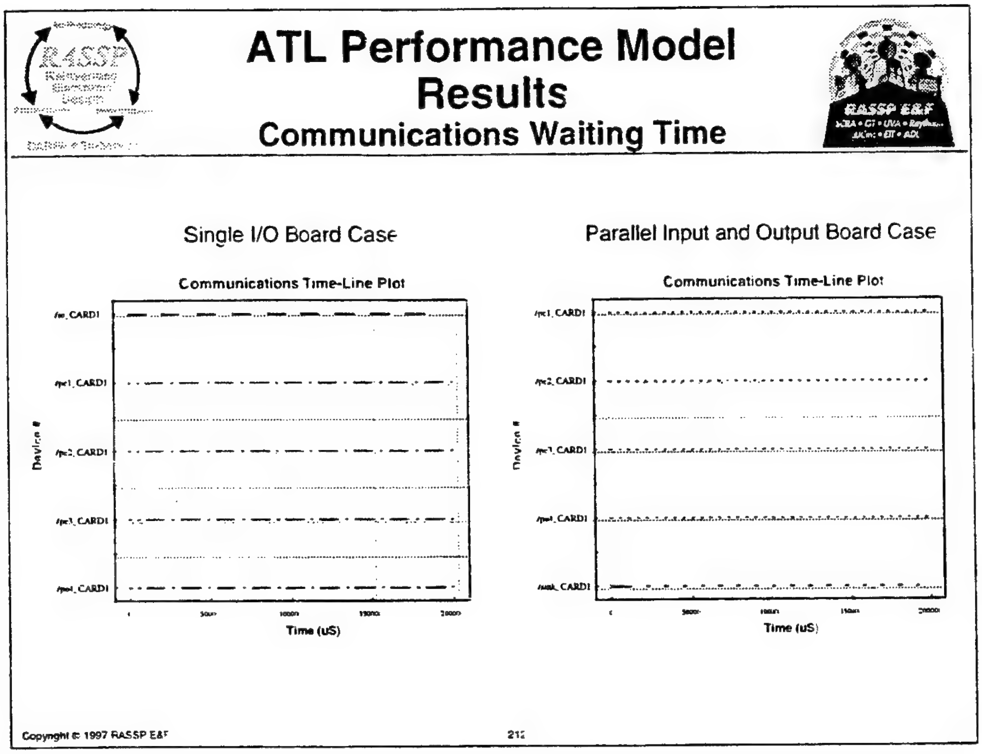
Single I/O Board Case

Processing Time-Line Plot

Parallel Input and Output Board Case

Processing Time-Line Plot

Here is the results of the performance model from the STL time line tool. These graphs show the compute times for the modules. Notice that the second architectural alternative (with the Source and Sink processors) has much better throughput in terms of the number of loop iterations (> 20) than the first architectural alternative (<11).



## ATL Performance Model Results

### Communications Waiting Time

Single I/O Board Case

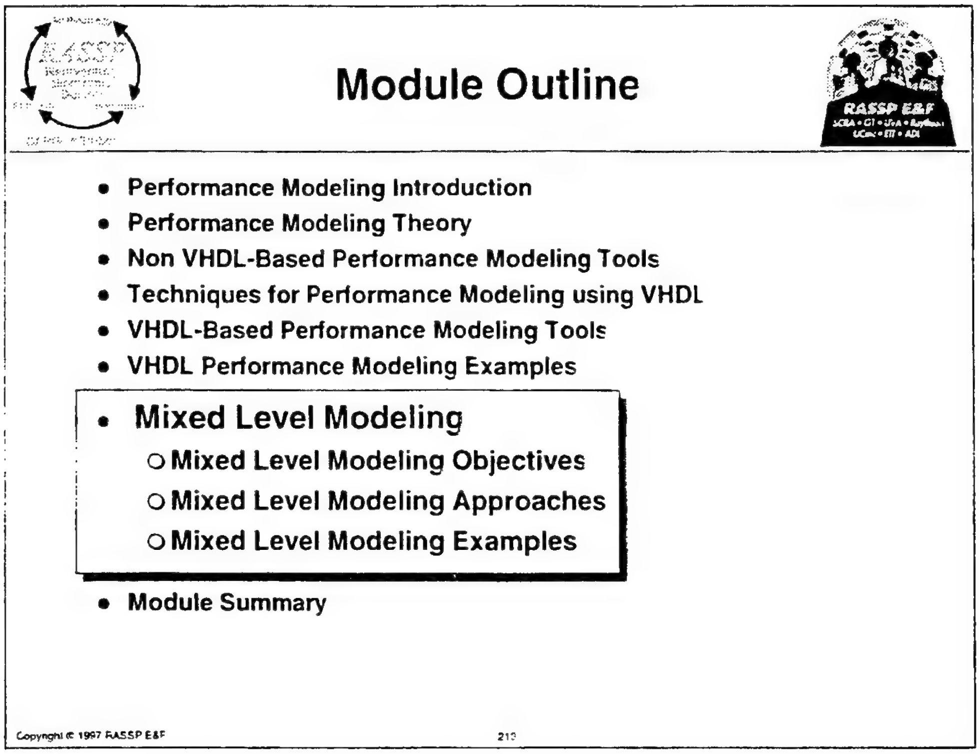
Communications Time-Line Plot

Parallel Input and Output Board Case

Communications Time-Line Plot

This is an activity time line plot of the communications (including waiting time) in the two alternative architectures. Note that the second architecture spends a great deal less time communicating or waiting for communications resulting in the higher throughput.

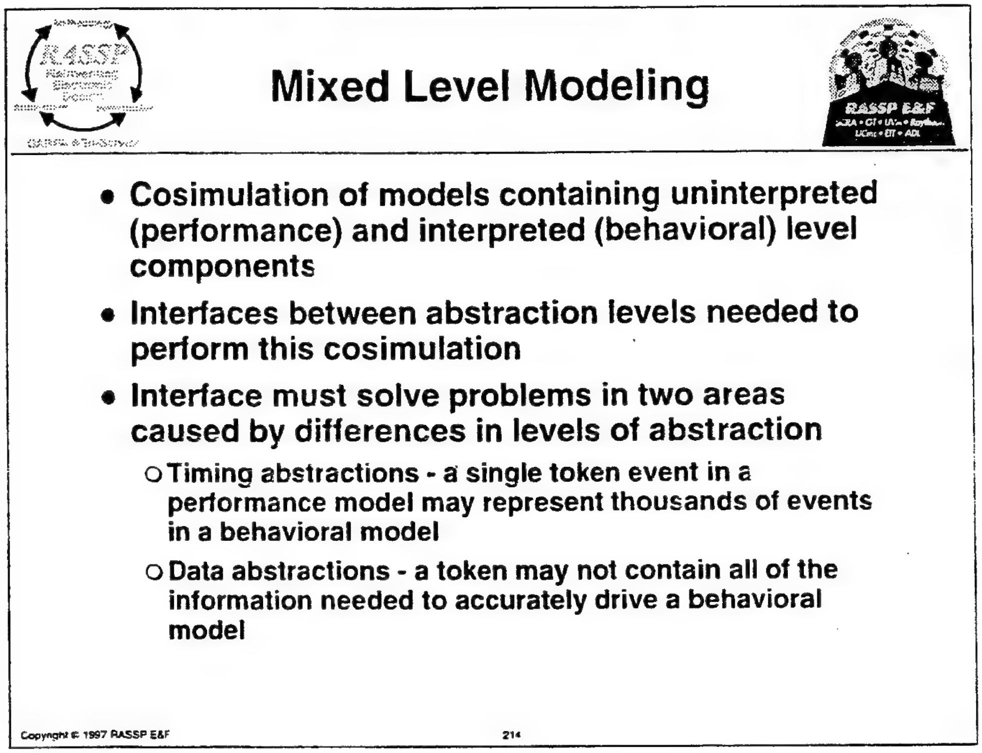
3!



## Module Outline

- Performance Modeling Introduction
- Performance Modeling Theory
- Non VHDL-Based Performance Modeling Tools
- Techniques for Performance Modeling using VHDL
- VHDL-Based Performance Modeling Tools
- VHDL Performance Modeling Examples
- **Mixed Level Modeling**
  - ○ Mixed Level Modeling Objectives
  - ○ Mixed Level Modeling Approaches
  - ○ Mixed Level Modeling Examples
- Module Summary

Module Outline



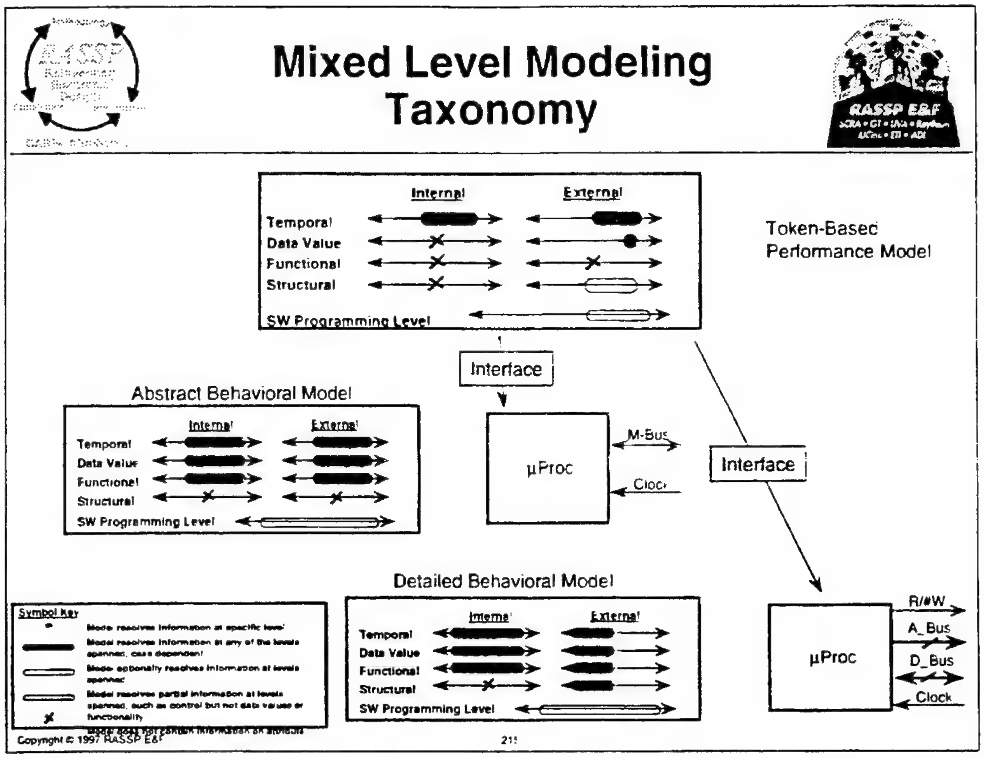
# Mixed Level Modeling

- **Cosimulation of models containing uninterpreted (performance) and interpreted (behavioral) level components**
- **Interfaces between abstraction levels needed to perform this cosimulation**
- **Interface must solve problems in two areas caused by differences in levels of abstraction**
    - **Timing abstractions - a single token event in a performance model may represent thousands of events in a behavioral model**
    - **Data abstractions - a token may not contain all of the information needed to accurately drive a behavioral model**

This section explains the concept of mixed level modeling, the cosimulation of performance and behavioral models, and how it is implemented in ADEPT. ADEPT was chosen as the example for this section as the theory and implementation of mixed level modeling is more advanced in ADEPT than other performance modeling environments as of this date. More information on this subject can be obtained from the UVa RASSP web page:

http://csis.ee.virginia.edu/~rassp

Cosmos (through PML) includes the capability for constructing mixed level models, but the facilities for developing methods to resolve timing and data abstraction are less well developed and require more user interaction.



## Mixed Level Modeling Taxonomy

| | Internal | External |
|---|---|---|
| Temporal | | |
| Data Value | | |
| Functional | | |
| Structural | | |
| SW Programming Level | | |

Token-Based Performance Model

Interface

Abstract Behavioral Model

| | Internal | External |
|---|---|---|
| Temporal | | |
| Data Value | | |
| Functional | | |
| Structural | | |
| SW Programming Level | | |

µProc

M-Bus

Clock

Interface

Detailed Behavioral Model

| | Internal | External |
|---|---|---|
| Temporal | | |
| Data Value | | |
| Functional | | |
| Structural | | |
| SW Programming Level | | |

µProc

R/#W

A_Bus

D_Bus

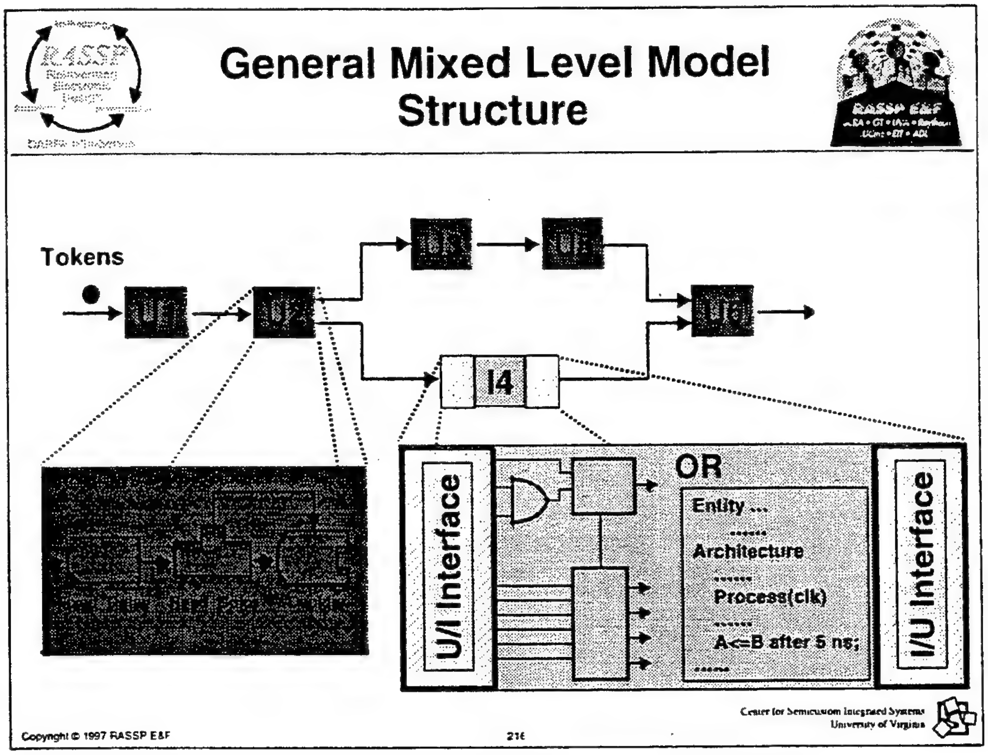
Clock

Symbol Key

- ● Model resolves information at specific level
- Model resolves information at any of the levels spanned, case dependent
- Model optionally resolves information at levels spanned
- Model resolves partial information at levels spanned, such as control but not data values or functionality
- ✗ Model does not contain information on attribute

215

This figure illustrates where the components of mixed level models lie in the RASSP taxonomy. It is clear from this description that token-based performance models have abstract timing and little or no data values (and data transformations - function) and that behavioral models have more detailed timing and data values. Therefore, it is easy to see that an interface(s) is needed between them when they are simulated in the same model.



## General Mixed Level Model Structure

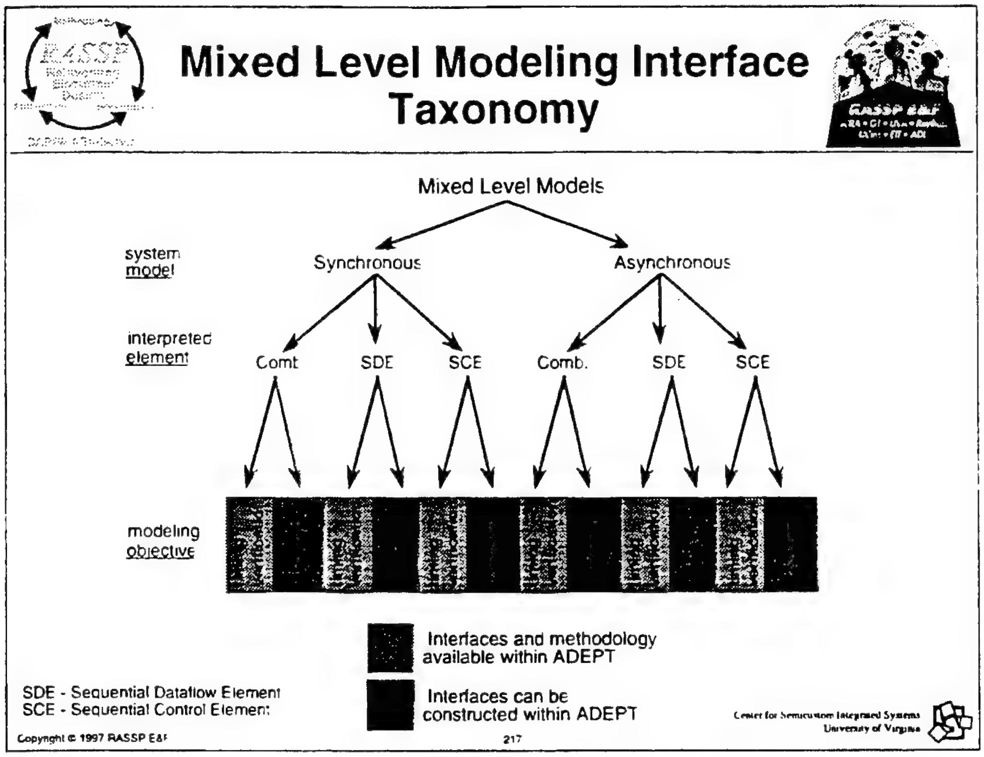
```
Entity ...
   ......
Architecture
   ......
   Process(clk)
   ......
   A<=B after 5 ns;
......
```

Copyright © 1997 RASSP E&F

216

Center for Semicustom Integrated Systems
University of Virginia

This is the general structure of a mixed level model. Here a single component in the performance model has been replaced with a behavioral component. Interfaces are required on its input and output to resolve the tokens to values and values to token conversion problem.



## Mixed Level Modeling Interface Taxonomy

Mixed Level Models

system model: Synchronous, Asynchronous

interpreted element: Comb., SDE, SCE, Comb., SDE, SCE

modeling objective

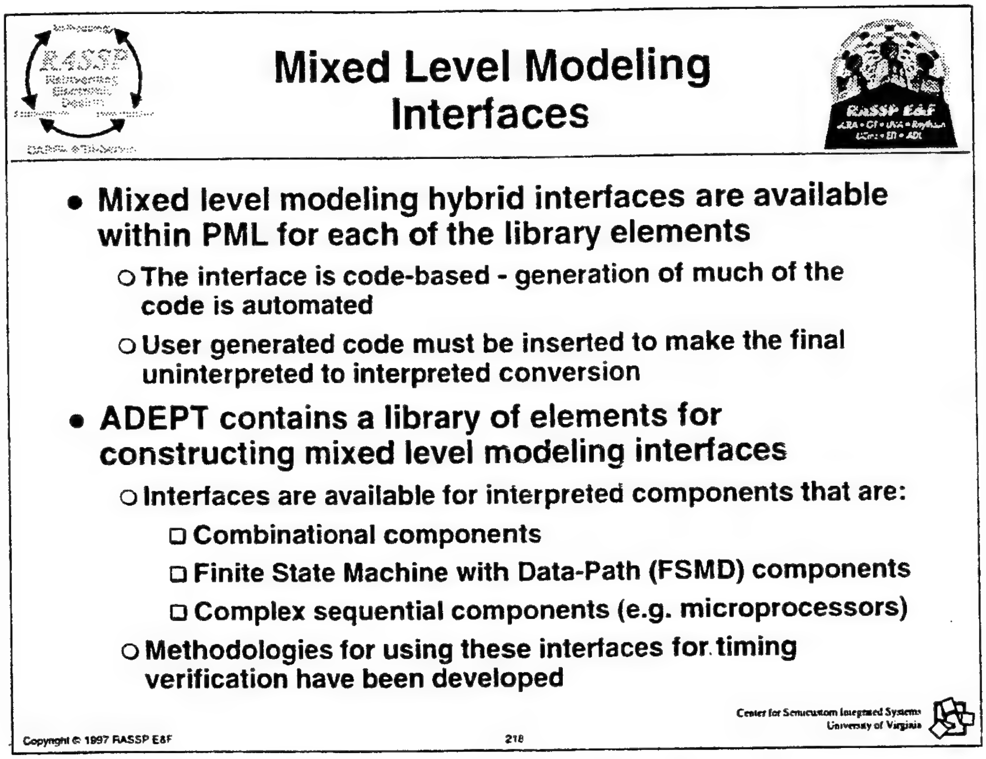
▨ Interfaces and methodology available within ADEPT

■ Interfaces can be constructed within ADEPT

SDE - Sequential Dataflow Element
SCE - Sequential Control Element

Center for Semicustom Integrated Systems
University of Virginia

This figure shows the taxonomy of hybrid models that was developed jointly between UVa and Honeywell Technology Center (their version is slightly different) to classify the solutions. Note that most work thus far has concentrated on the problem of timing verification.



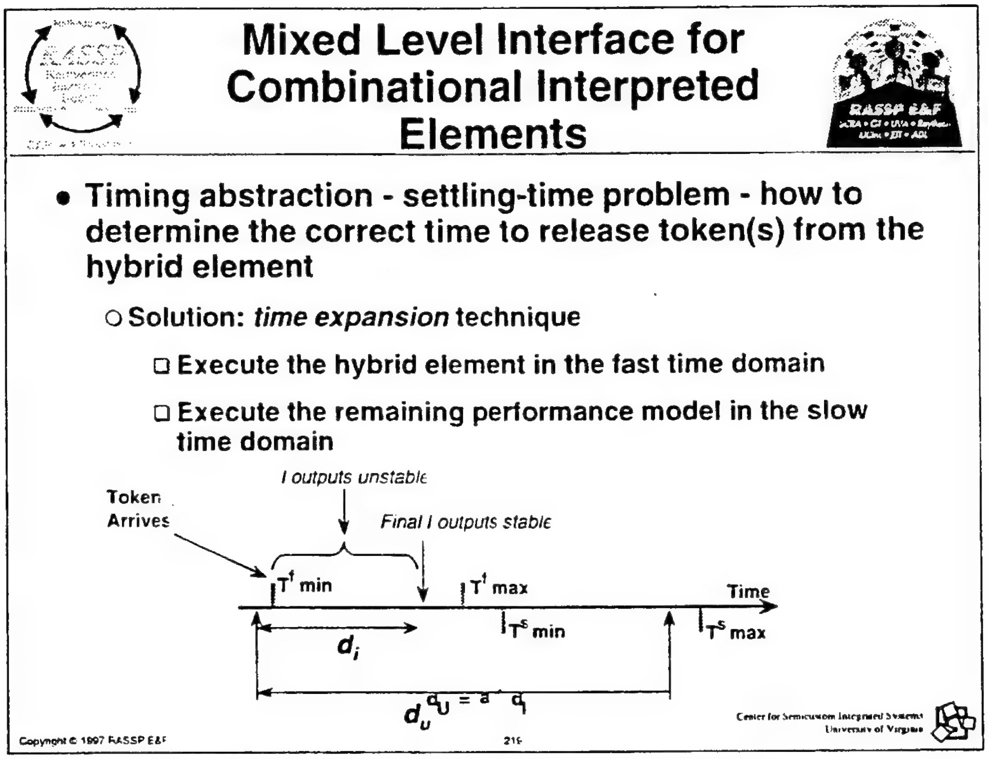
## Mixed Level Modeling Interfaces

- **Mixed level modeling hybrid interfaces are available within PML for each of the library elements**
  - ○ The interface is code-based - generation of much of the code is automated
  - ○ User generated code must be inserted to make the final uninterpreted to interpreted conversion
- **ADEPT contains a library of elements for constructing mixed level modeling interfaces**
  - ○ Interfaces are available for interpreted components that are:
    - ☐ Combinational components
    - ☐ Finite State Machine with Data-Path (FSMD) components
    - ☐ Complex sequential components (e.g. microprocessors)
  - ○ Methodologies for using these interfaces for timing verification have been developed

Center for Semicustom Integrated Systems
University of Virginia

As stated previously, PML has a mixed level modeling interface capability, but it is mainly code based and the user must supply the VHDL code that performs the tokens to values and values to tokens conversion.

ADEPT has a library of standard "hybrid" elements out of which mixed level modeling interfaces can be developed. For some classes of models in the taxonomy, the interface can be generated with no user coding, or new modules required. In other cases, some generation of application specific modules by the user is required.

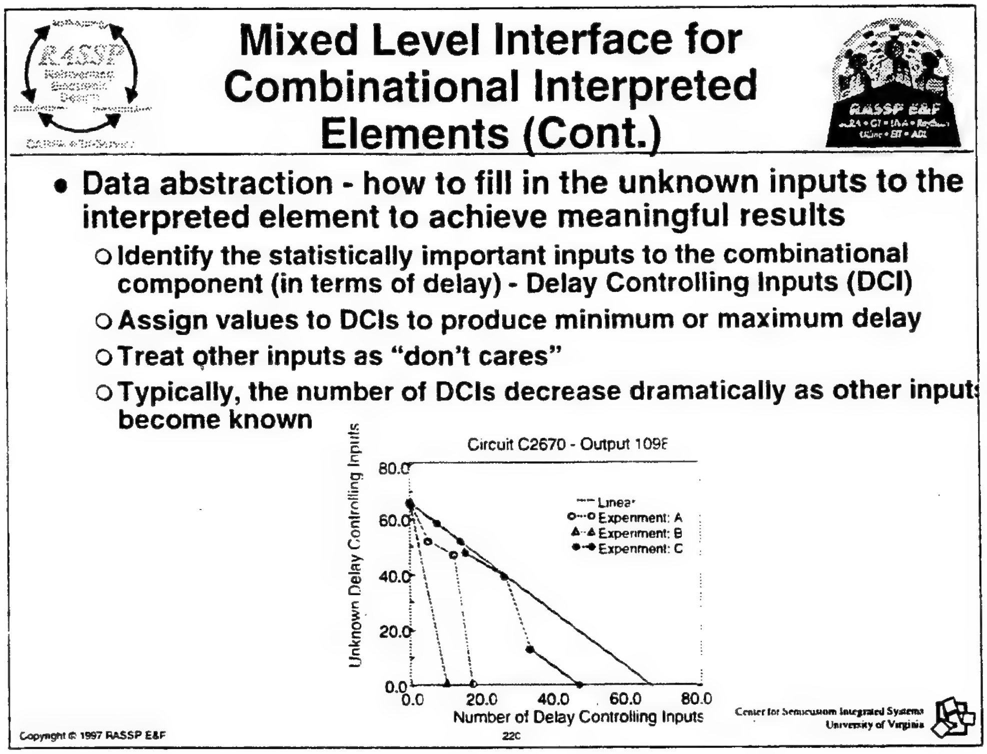


## Mixed Level Interface for Combinational Interpreted Elements

- **Timing abstraction - settling-time problem - how to determine the correct time to release token(s) from the hybrid element**
  - **Solution: *time expansion* technique**
    - **Execute the hybrid element in the fast time domain**
    - **Execute the remaining performance model in the slow time domain**

Center for Semicustom Integrated Systems
University of Virginia

This figure illustrates the problem of timing in mixed level models when the behavioral (or interpreted) element is combinational. A token arriving at the interface to the hybrid element, which contains the interpreted combinational component, triggers application of the new values to the inputs of the combinational element. Then after some time, the generation of the final outputs from the combinational element will trigger the release of the token from the hybrid element. The problem is the fact that the outputs of the combinational element take variable times to settle to the final value and it is difficult to determine when that has happened. The solution, called time expansion, is to run the combinational element in "fast time" which is usually 10 times faster than the performance model time scale, wait the maximum delay time of the combinational element in fast time, observe when, in fast time, the combinational element's outputs settled to their final values, and then scale this time up to slow time and release the token at the proper time in slow time.



# Mixed Level Interface for Combinational Interpreted Elements (Cont.)

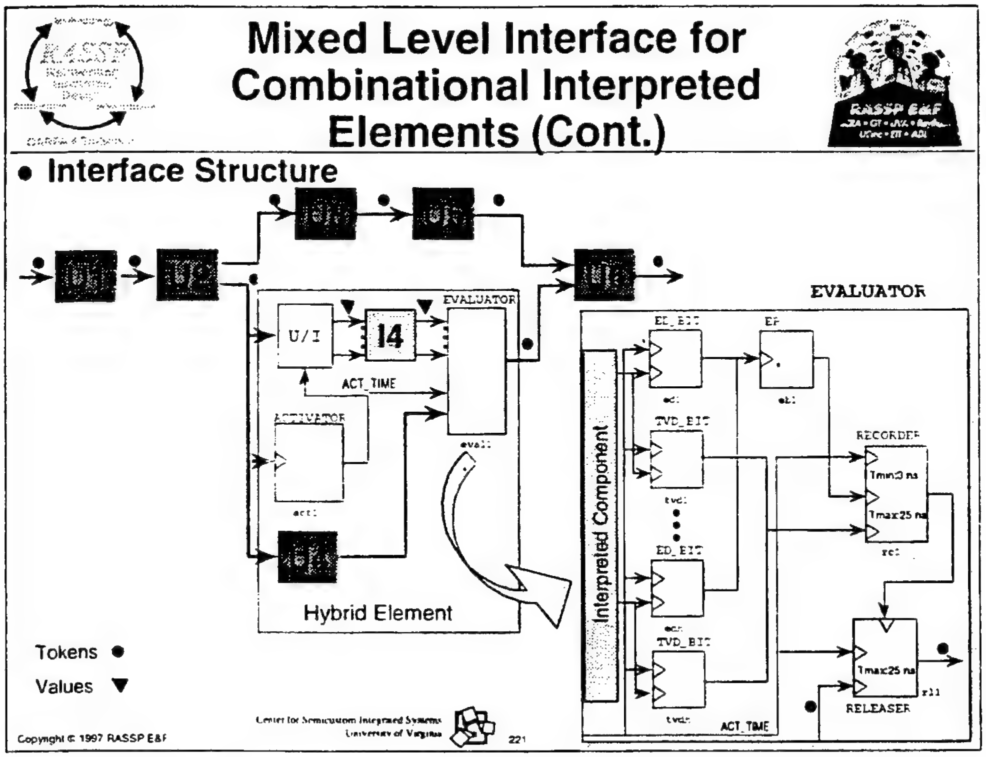
- Data abstraction - how to fill in the unknown inputs to the interpreted element to achieve meaningful results
  - Identify the statistically important inputs to the combinational component (in terms of delay) - Delay Controlling Inputs (DCI)
  - Assign values to DCIs to produce minimum or maximum delay
  - Treat other inputs as "don't cares"
  - Typically, the number of DCIs decrease dramatically as other input become known

22C

Center for Semicustom Integrated Systems
University of Virginia

Another problem attacked in the mixed level area in ADEPT is the problem of specifying the inputs to the combinational element, that could not be derived from the incoming toke (called "unknown inputs") in such a way as to generate meaningful results, usually either minimum or maximum delay.

A technique has been developed to determine the inputs that have the most influence on the delay of the combinational element (called DCIs) and setting them to the values that cause the best or worst case values. In theory, this is an exponentially complex problem, but the results, as shown here, have demonstrated that as a few inputs are known from the performance model, the number of DCIs drops dramatically, resulting in the problem quickly becoming tractable.



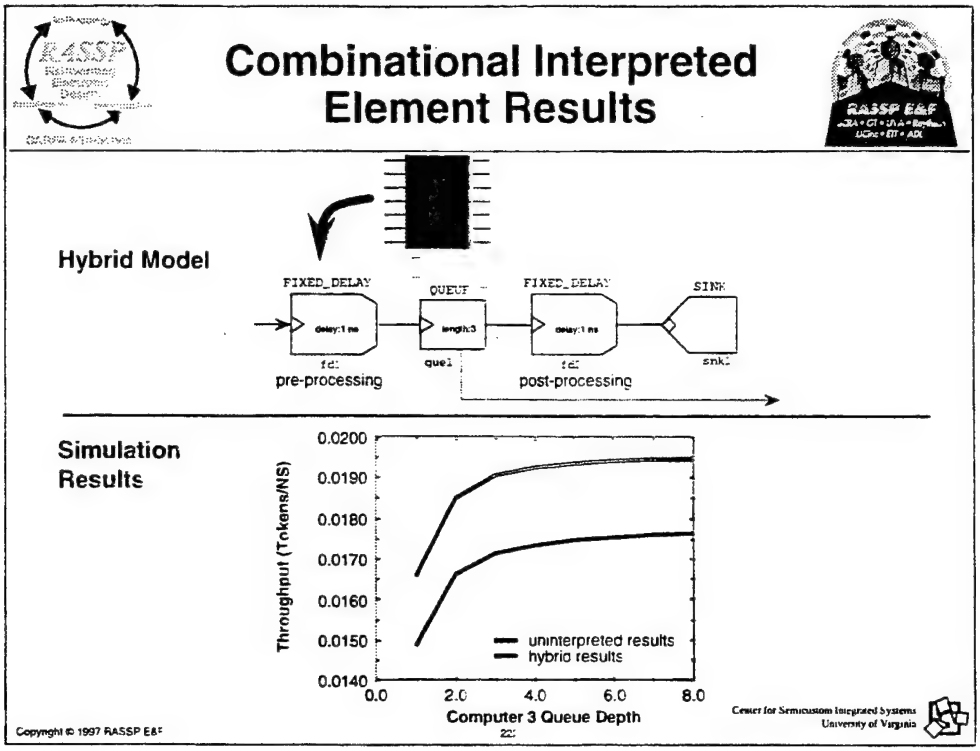
## Mixed Level Interface for Combinational Interpreted Elements (Cont.)

- Interface Structure

Center for Semicustom Integrated Systems
University of Virginia

221

This is the structure of the mixed level interface in ADEPT for combinational interpreted elements that implements time expansion. When a token arrives at the input to the hybrid element, the U/I component converts values on the token to values on the combinational element's inputs and runs the DCI algorithm if need be. At the same time, the activator records the token arrival time and passes it to the evaluator. The evaluator waits the maximum combination delay time in fast time, measures the actual combination delay in fast time, and scales that up and releases the token at the proper time in slow time.



## Combinational Interpreted Element Results

**Hybrid Model**

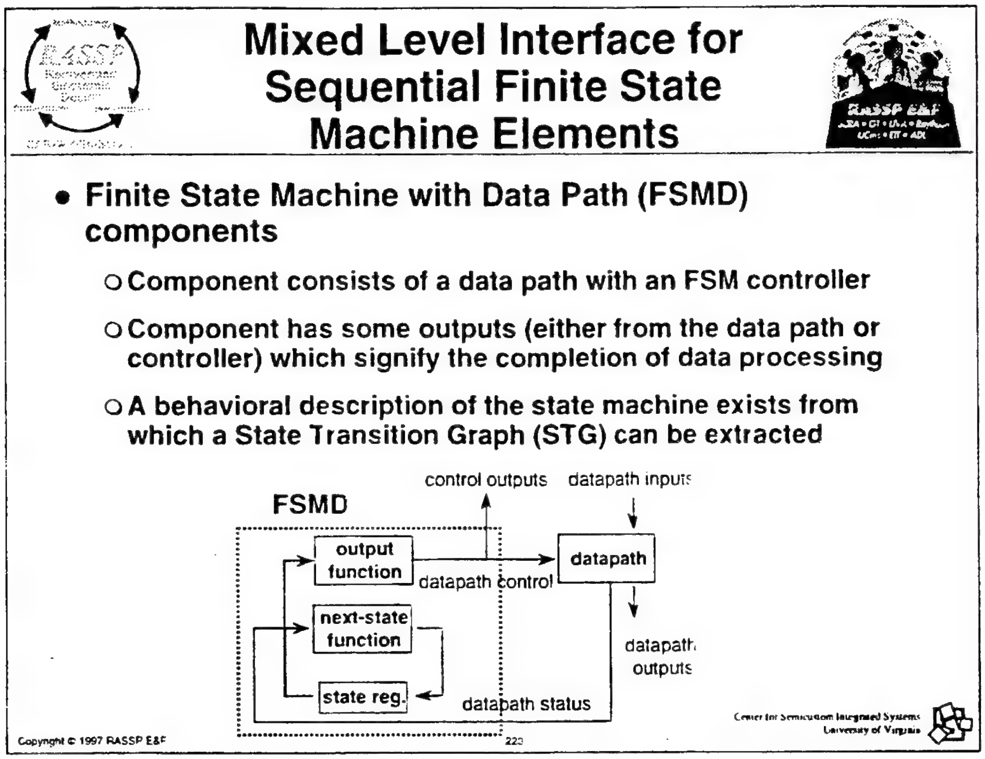
FIXED_DELAY
delay:1 ns
fd1
pre-processing
QUEUE
length:3
que1
FIXED_DELAY
delay:1 ns
fd2
post-processing
SINK
snk1

**Simulation Results**

Throughput (Tokens/NS)
0.0200
0.0190
0.0180
0.0170
0.0160
0.0150
0.0140
0.0
2.0
4.0
6.0
8.0
Computer 3 Queue Depth
uninterpreted results
hybrid results

Center for Semicustom Integrated Systems
University of Virginia

Here are some simple results from a mixed level model with a combinational element. Note that the throughput achieved by the mixed level model has the same shape as the original performance modeling results (which is good), but it is shifted as a result of having actual delay values from the behavioral component.



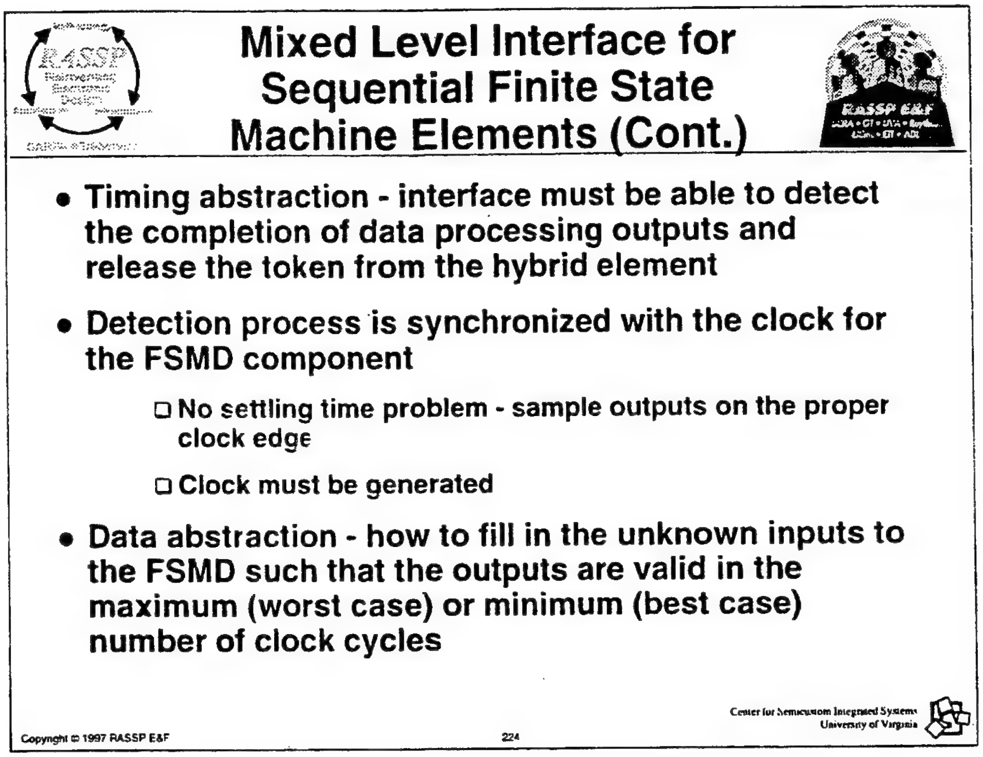
## Mixed Level Interface for Sequential Finite State Machine Elements

- **Finite State Machine with Data Path (FSMD) components**
  - ○ Component consists of a data path with an FSM controller
  - ○ Component has some outputs (either from the data path or controller) which signify the completion of data processing
  - ○ A behavioral description of the state machine exists from which a State Transition Graph (STG) can be extracted

Center for Semiconductor Integrated Systems
University of Virginia

Another area of mixed level modeling investigated in ADEPT was that of an interpreted component that was a finite state machine with datapath (FSMD). This is an interpreted component who's function can be described by a state transition graph (STG). This is important because it allows graph algorithms to be used to analyze the STG to determine maximum and minimum delay. In addition, a requirement is that there be some outputs, either from the state machine or datapath, the can be used to determine the completion of processing for a given token arrival event.

Examples of these types of elements include a dedicated FFT chip or a floating point coprocessor.



## Mixed Level Interface for Sequential Finite State Machine Elements (Cont.)

- Timing abstraction - interface must be able to detect the completion of data processing outputs and release the token from the hybrid element
- Detection process is synchronized with the clock for the FSMD component
  - ☐ No settling time problem - sample outputs on the proper clock edge
  - ☐ Clock must be generated
- Data abstraction - how to fill in the unknown inputs to the FSMD such that the outputs are valid in the maximum (worst case) or minimum (best case) number of clock cycles

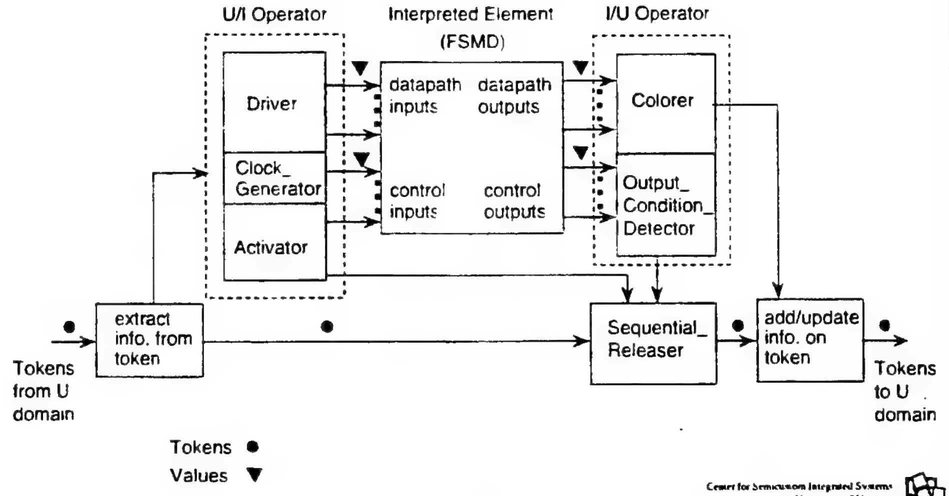
Center for Semicustom Integrated Systems
University of Virginia

The timing abstraction problem is easier with FSMD components as there is no settling problem - everything is resolved on a clock edge. However, the clock input to the FSMD must be generated, usually by the mixed level interface elements.

The data abstraction problem is similar to the combinational element one - how to specify the unknown inputs such that minimum or maximum delay results.

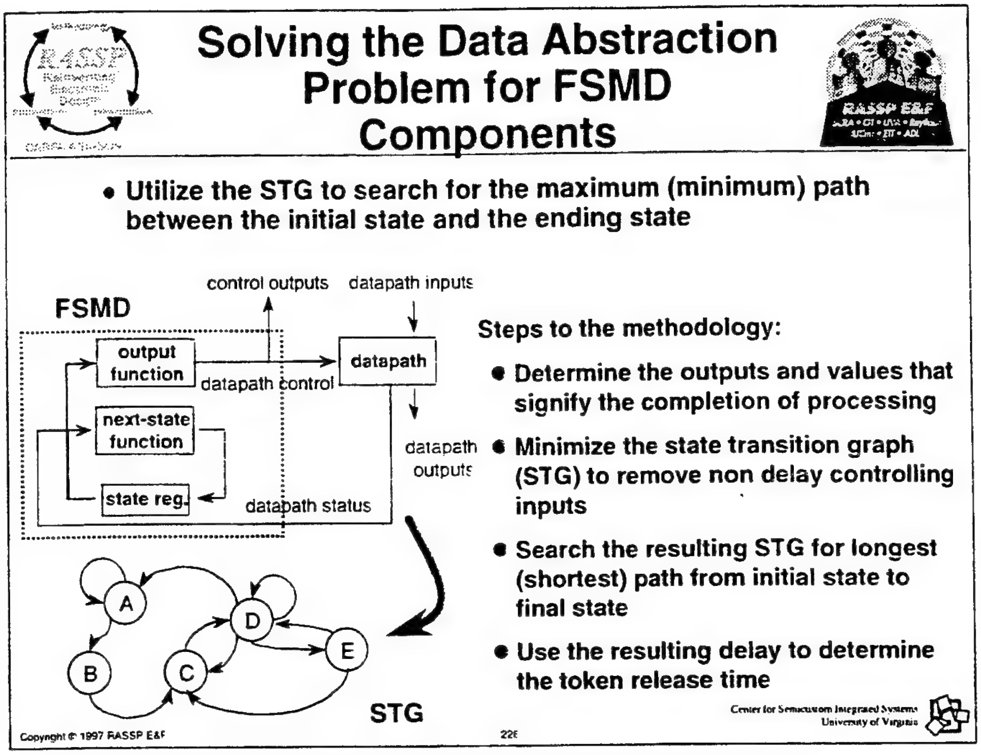
# Mixed Level Interface for Sequential Finite State Machine Elements (Cont.)

- **Interface Structure**



Center for Semiconductor Integrated Systems
University of Virginia

This is the structure of the mixed level interface for FSMD components. The driver and clock generator perform the U/I function and the activator performs the same function as in the previous example. The Colorer, output_condition_detector, and sequential_releaser perform the functions of the evaluator, that is, determining when to release the token from the hybrid element after the proper delay time according to the interpreted component.



## Solving the Data Abstraction Problem for FSMD Components

- **Utilize the STG to search for the maximum (minimum) path between the initial state and the ending state**

**Steps to the methodology:**

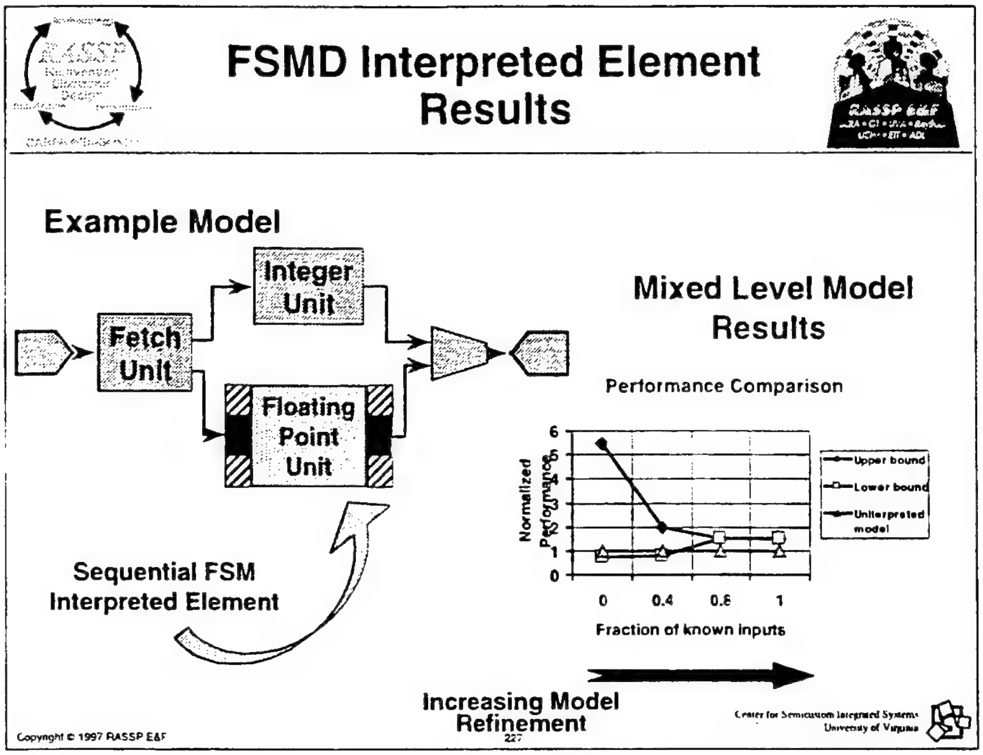
- **Determine the outputs and values that signify the completion of processing**
- **Minimize the state transition graph (STG) to remove non delay controlling inputs**
- **Search the resulting STG for longest (shortest) path from initial state to final state**
- **Use the resulting delay to determine the token release time**

Center for Semicustom Integrated Systems University of Virginia

Copyright © 1997 RASSP E&F

226

This figure outlines the methodology used to determine the minimum or maximum delay, in terms of clock cycles, for the FSMD interpreted component using the component's STG. First, the outputs that do not affect when the token is released are removed from the STG and the resulting STG is simplified. Next, the resulting STG is searched to find the shortest (minimum time) or longest (maximum time) path from the initial state to the final state. Finally, the inputs necessary to drive the FSMD along this path are applied to the interpreted component in the simulation.



## FSMD Interpreted Element Results

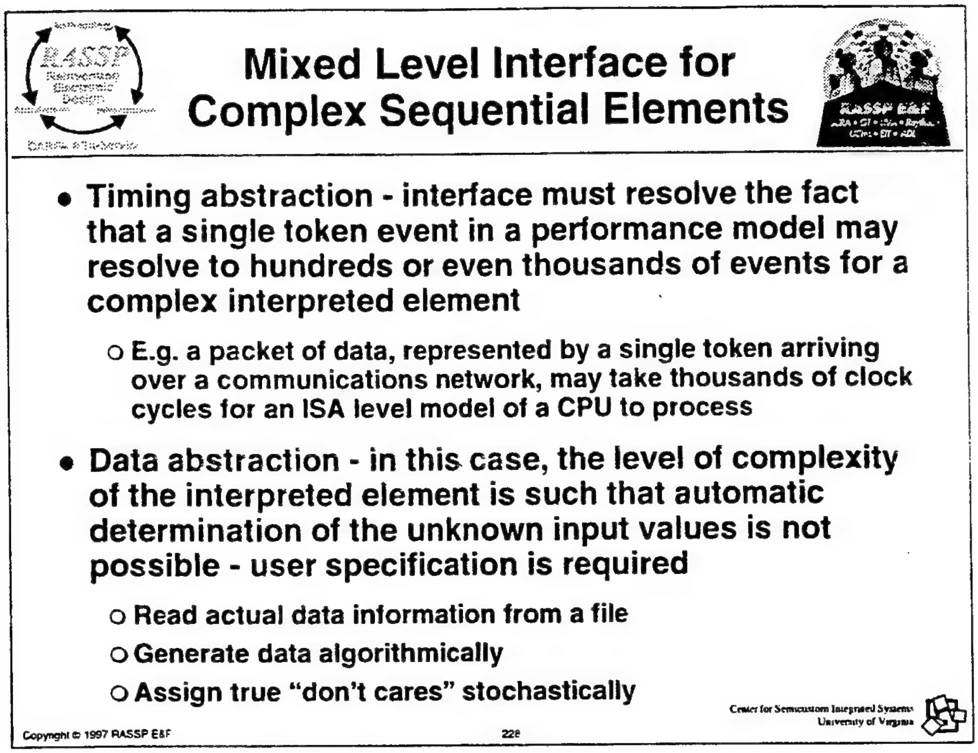
**Example Model**

**Sequential FSM Interpreted Element**

**Mixed Level Model Results**

**Increasing Model Refinement**

Here are some results from an example of applying the technique to an FSMD mixed level model. In this case, it was a performance model of a processor with a fetch unit, an integer unit and a floating point unit. The floating point unit was replaced with its interpreted (behavioral) representation. The results show how the upper and lower bounds (minimum and maximum delay) on performance can be generated for the model at various levels of refinement. As the model is refined, the fraction of inputs for which the actual values are known from the performance model increase, and the bounds get tighter and finally converge. Also notice that, as is quite typical, the initial estimate of the performance as used in the high level performance model, was inaccurate.



## Mixed Level Interface for Complex Sequential Elements

- **Timing abstraction - interface must resolve the fact that a single token event in a performance model may resolve to hundreds or even thousands of events for a complex interpreted element**
    - ○ E.g. a packet of data, represented by a single token arriving over a communications network, may take thousands of clock cycles for an ISA level model of a CPU to process
- **Data abstraction - in this case, the level of complexity of the interpreted element is such that automatic determination of the unknown input values is not possible - user specification is required**
    - ○ Read actual data information from a file
    - ○ Generate data algorithmically
    - ○ Assign true "don't cares" stochastically

Copyright © 1997 RASSP E&F

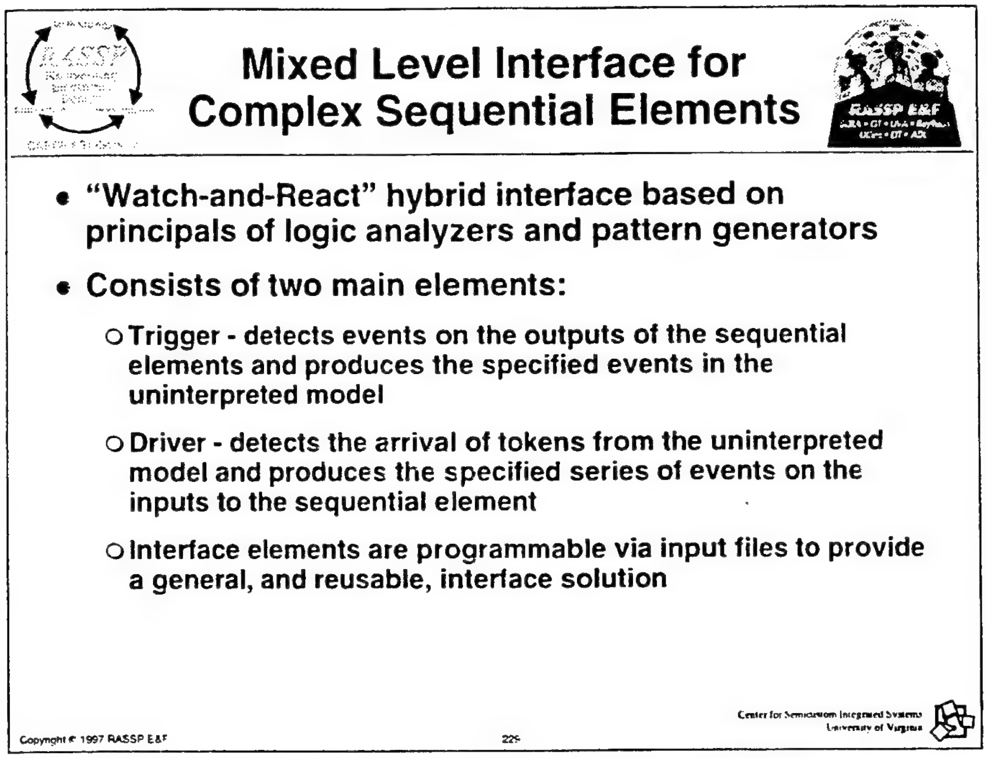
228

Center for Semicustom Integrated Systems
University of Virginia

Finally, mixed level interface elements were developed for "complex sequential elements" which are sequential elements that are too complex to describe as state machines. In this case, the interface is more ad hoc, and is targeted at solving the timing abstraction problem. The user must solve the data abstraction problem for interpreted elements such as these.

Elements that fall into this category include microprocessors, microcontrollers, and even entire computer systems.

3



## Mixed Level Interface for Complex Sequential Elements

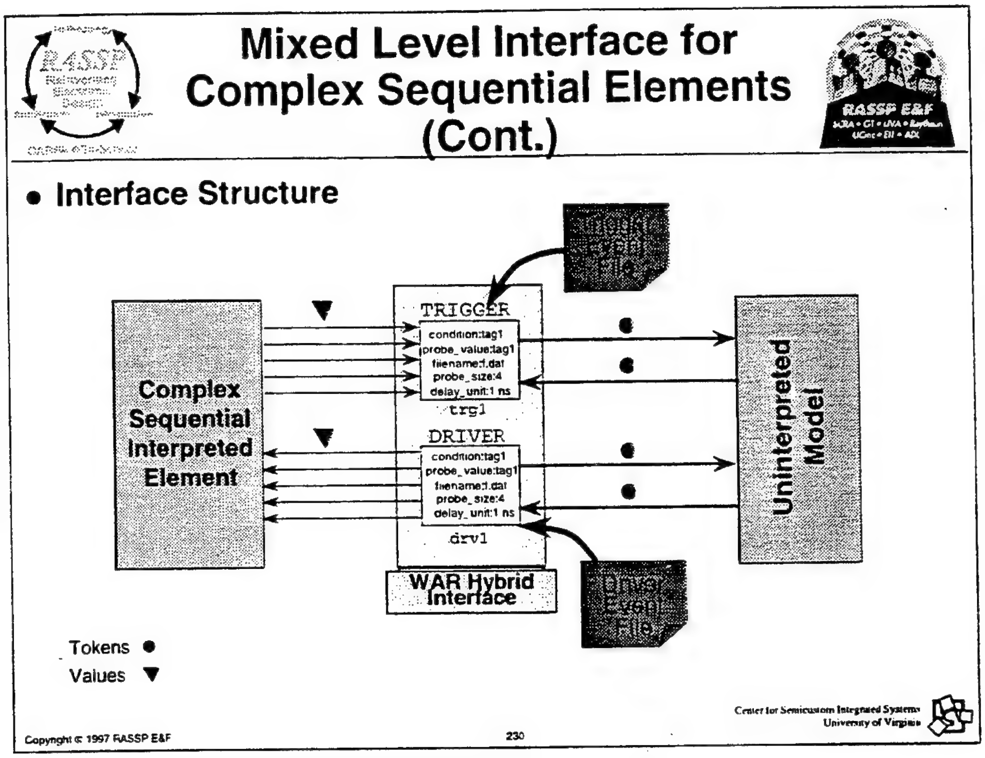
- "Watch-and-React" hybrid interface based on principals of logic analyzers and pattern generators
- Consists of two main elements:
  - Trigger - detects events on the outputs of the sequential elements and produces the specified events in the uninterpreted model
  - Driver - detects the arrival of tokens from the uninterpreted model and produces the specified series of events on the inputs to the sequential element
  - Interface elements are programmable via input files to provide a general, and reusable, interface solution

Copyright © 1997 RASSP E&F

229

Center for Semicustom Integrated Systems
University of Virginia

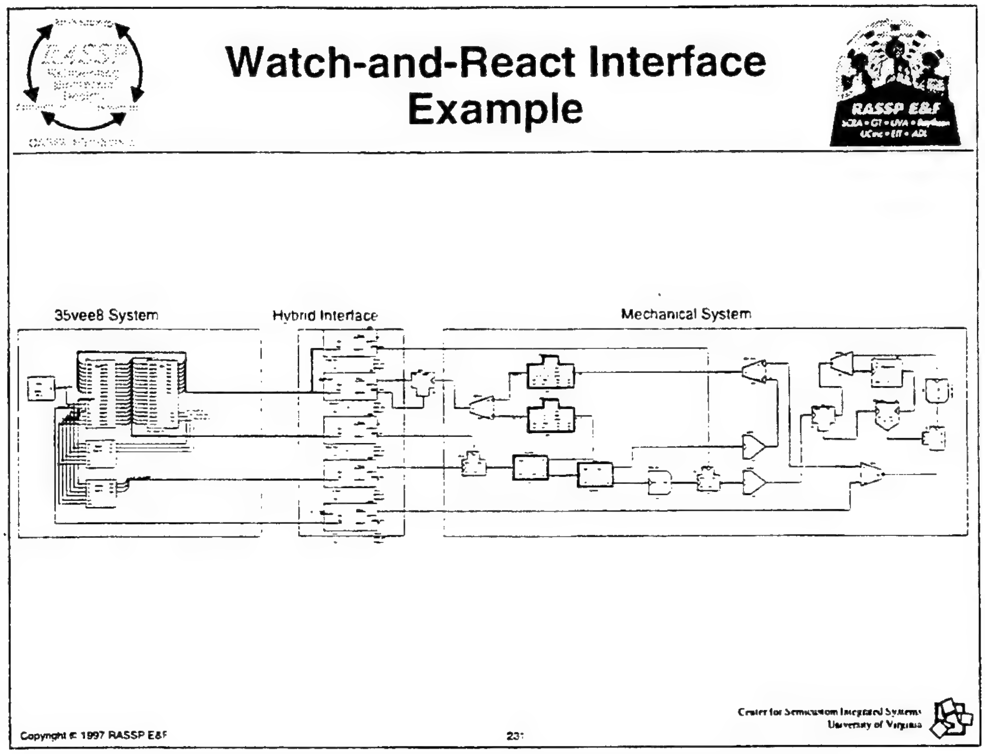
The so called "watch and react" hybrid interface is build on the principals of logic analyzers. The interface watches the outputs of the interpreted element for certain "trigger" conditions, and when they occur, it takes the appropriate action. Likewise, when the performance model dictates that some new inputs be supplied to the interpreted component, a "program" can be executed that generates a complex set of input sequences to the interpreted component.



# Mixed Level Interface for Complex Sequential Elements (Cont.)

- **Interface Structure**

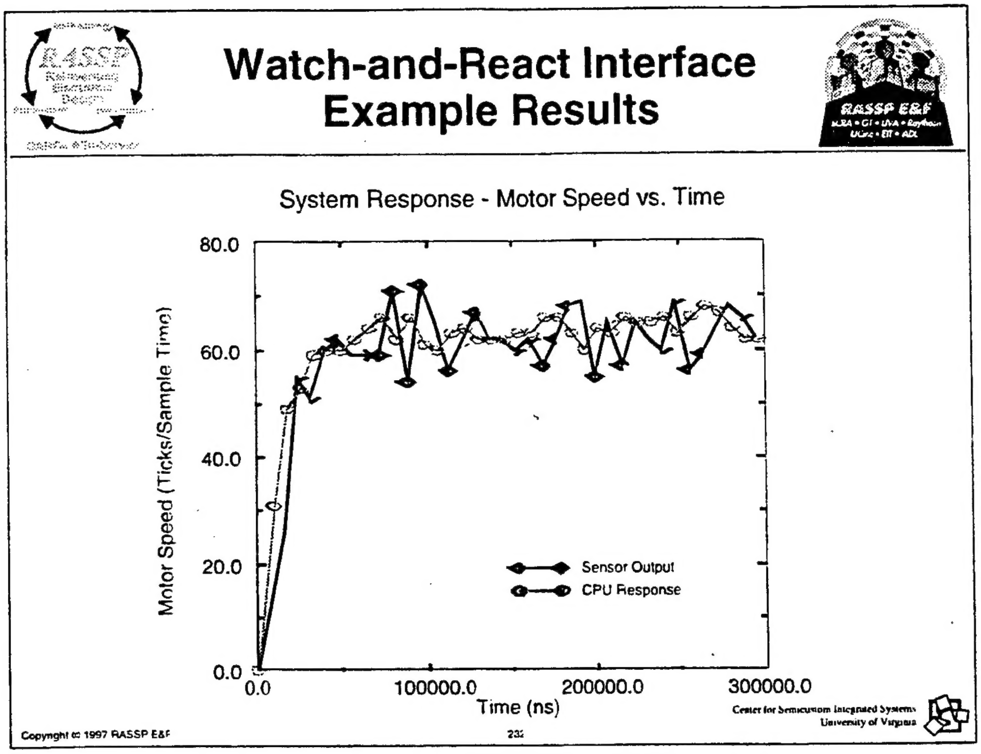
Here is the general structure of the watch and react interface. Both the trigger and driver element can be programmed by input files - which keeps them general in nature and avoids having the user to generate new VHDL code for a specific application.



## Watch-and-React Interface Example

Center for Semicustom Integrated Systems
University of Virginia

Here is an example of the use of the watch and react interface. It is a motor control system in which an actual behavioral model of a microcontroller, along with its associated memory system has been inserted. The motor control system and its feedback mechanism are modeled at a system level using ADEPT modules.

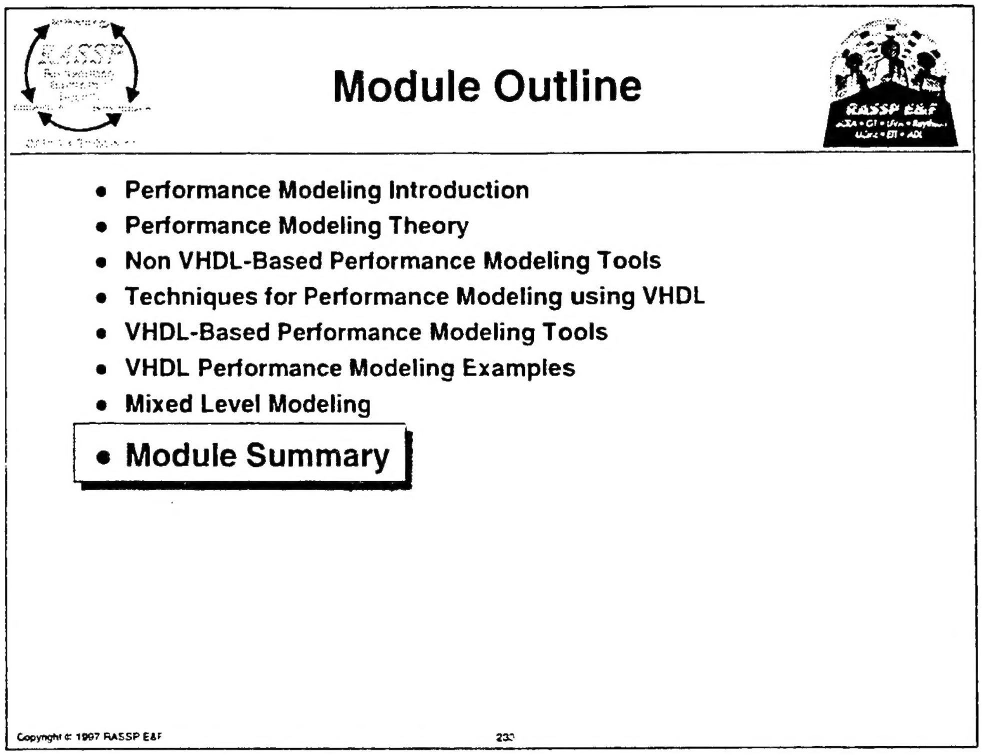


## Watch-and-React Interface Example Results

System Response - Motor Speed vs. Time

Center for Semicustom Integrated Systems
University of Virginia

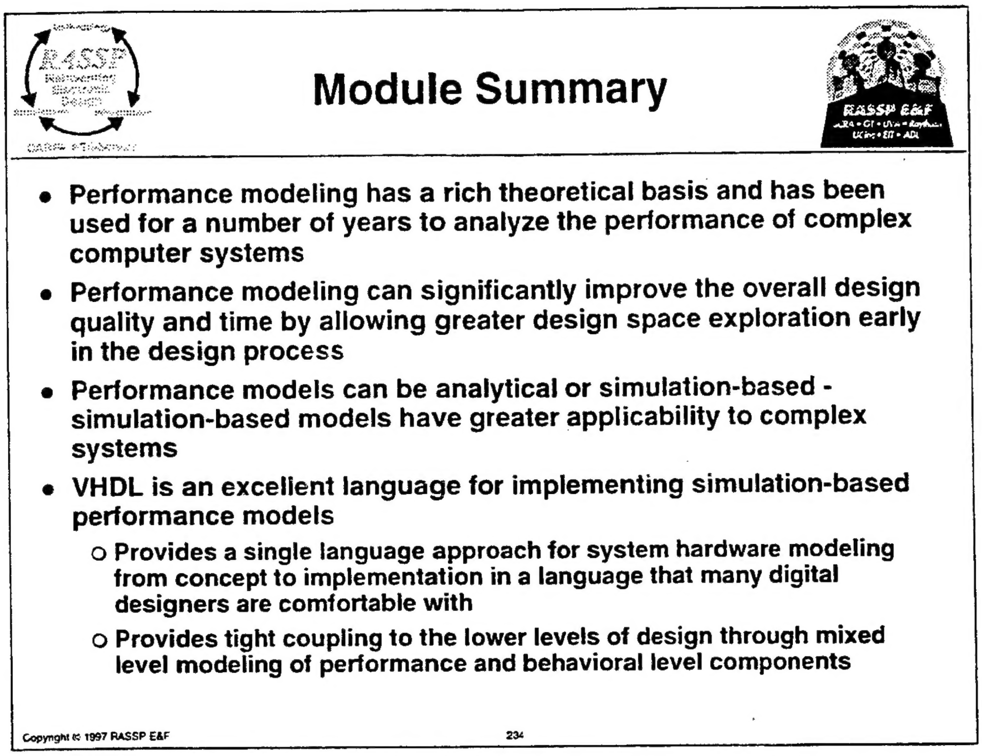
Here are some results from the mixed level model illustrating the proper control of the motor speed. Note that the behavioral model of the microcontroller is executing an actual control program from a memory model and responding to perturbations in the motor speed in the performance model of the motor system.



## Module Outline

- Performance Modeling Introduction
- Performance Modeling Theory
- Non VHDL-Based Performance Modeling Tools
- Techniques for Performance Modeling using VHDL
- VHDL-Based Performance Modeling Tools
- VHDL Performance Modeling Examples
- Mixed Level Modeling
- **Module Summary**

Module Outline



# Module Summary

- Performance modeling has a rich theoretical basis and has been used for a number of years to analyze the performance of complex computer systems
- Performance modeling can significantly improve the overall design quality and time by allowing greater design space exploration early in the design process
- Performance models can be analytical or simulation-based - simulation-based models have greater applicability to complex systems
- VHDL is an excellent language for implementing simulation-based performance models
    - Provides a single language approach for system hardware modeling from concept to implementation in a language that many digital designers are comfortable with
    - Provides tight coupling to the lower levels of design through mixed level modeling of performance and behavioral level components

# References

[Jain91] Jain, R., The Art of Computer Systems Performance Analysis. Techniques for Experimental Design, Measurement, and Modeling, John Wiley & Sons, Inc., 1991.

[Hein96] Hein, C., T. Carpenter, "Tutorial: VHDL-Based Rapid Prototyping for Large DSP Systems," Presented at the Second Annual RASSP Conference, October 10th, 1996.

[Hein97] Hein, C., T. Carpenter, A. Gadient, R. Harr, P. Kalutkiewicz, V. Madisetti, "RASSP VHDL Modeling Terminology and Taxonomy," Revision 2.2, March 27, 1997.

[Sauer81] Sauer, C. H., K. M. Chandy, Computer Systems Performance Modeling, Prentice-Hall, Inc., 1981.

[Kant92] Kant, K., Introduction to Computer System Performance Evaluation,.McGraw-Hill, Inc., 1992

[Murata89] Murata, T., "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, Vol. 77, No. 4, April 1989

[Cassandras93] Cassandras, Christos G., Discrete Event Systems, Modeling and Performance Analysis, Aksen Associates Incorporated Publishers, 1993.

[Ptolemy 96] Lee, E. A., et. al., The Almagest Volumes 1-4. - The Ptolemy Reference Manual, 1996.

[Pauer97] Pauer, E. K., "High Performance Scalable Computing Performance Modeling Using Ptolemy" Proceedings of the IASTED International Conference on Modeling and Simulation, May 1997, pp 452-455

[ADEPT_LR96] ADEPT A.1 Library Reference Manual, CSIS Technical Report No. 960625, Department of Electrical Engineering, University of Virginia, December, 1996.

[ADEPT_UM96] Unified Modeling Reference Manual (ADEPT Version A.1), CSIS Technical Report No 960620.0, Department of Electrical Engineering, University of Virginia, December, 1996

[HTC97] RASSP VHDL Performance Modeling Interoperability Guideline, Version 3.0, Honeywell Technology Center, March 31, 1997.